UNIVERSITY OF OSLO
Department of Informatics

Effective replication of large data in distributed hash tables

Ali Amin Ahmad

Spring 2013

Effective replication of large data in distributed hash tables

Ali Amin Ahmad

Spring 2013

Abstract

In recent years several peer-to-peer systems have been launched. Among them are, Spotify [20], Skype [36] and BitCoin [28]. Another interesting and developing research field in peer-to-peer systems is online social networks such as [9, 16]. Peer-to-peer systems can autonomously organise themselves to provide DHT functionality. Distributed hash tables, known as DHTs, offer a key-value storage service. DHTs have been used in many storage services [10, 34, 42] in addition to [9, 16]. Although DHTs are very efficient way of storing and looking up small values or files, a problem arises when storing or updating relatively large data in DHTs. As values are often replicated, propagating updates to all replicas might be slow. The same problem occurs as peers leave the system and replicas must be created to serve users quickly or keep data available.

Peers in such systems may not have the necessary resources to propagate relatively large files quickly. Factors such as slow-senders or slow-receivers may delay the propagation and decrease the performance of such system. The delay can be multiplied in cases where slow-senders have to propagate large files to slow-receivers. Propagation delays aside, slow-senders must utilise their entire bandwidth to propagate such files. This creates a hotspot in the network and an obvious load imbalance, especially in a dynamic environment where replicas are created often.

In this thesis, we will propose a new propagation protocol for DHTs. Using this protocol, the system will be able to propagate relatively large data quickly while remaining load-balanced and keeping the overhead low. In addition, factors such as low-bandwidth connections and dynamic environments are taken into account. Our evaluation shows that our protocol can decrease the propagation time while introducing less than 1% overhead in most cases.

Preface

This thesis is based on the PhD Dissertation of Guido Urdaneta on Collaborative Wikipedia hosting [41]. This work would have required much more time and effort without using Guido's simulator.

I would like to thank Roman Vitenberg for his insightful questions, guidance and help during my thesis. The courses taught by Roman and his supervision of my thesis has helped improve, or possibly create, my critical way of thinking during my thesis and helped me think as a scientist.

I would like to thank Guido Urdaneta for sitting down with me and explaining how the simulator was build and how it could be used. His insight into the simulator and his guidance and supervision during the design, implementation and testing phases of the protocol were essential to completing this thesis. I will always remember that "Fixing compiler errors is easy, but the worst errors are the ones the compiler can not identify." I found that to be true based on my experience. Thanks to Abhishek Singh for providing a synthetic churn trace generator.

Thanks to the University in Oslo for giving me access to the Abel, the university's high-performance cluster. Without access to Abel, i would have to run simulations on my laptop. I would like to thank the university for providing a lab-space for it's master students. Thanks to my lab-mates Henning Klevjer, Kristoffer Jensen, Magnus Evensberget and Kent Are Varmedal as they made the long and exhausting days in the lab tolerable and even fun.

I am most thankful for my family and friends for being patient and motivating me through the thesis, especially my father for buying me a Visual Basic book that got me interested in Informatics and programming at an early age. To this day, i believe that book made a lot of difference in my life.

Contents

List of Figures

List of Tables

Part I

Background

Chapter 1

Introduction

1.1 The rise of peer-to-peer systems

In recent years a new era of peer-to-peer systems have emerged. According to Cisco visual networking index [1], peer-to-peer traffic will triple by 2016, as shown in Figure 1.1.

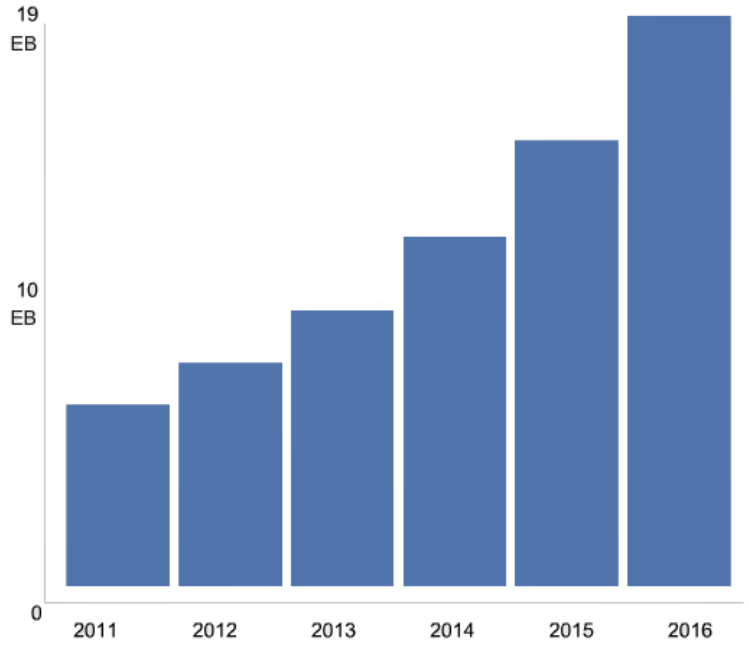

Figure 1.1: Peer-to-peer traffic projection

There has been a series of new peer-to-peer applications launched. Among them are Skype [36] , Spotify [37], BitTorrent [8], Bitcoin [28] etc.

Generally, there are two types of peer-to-peer systems; structured and unstructured. The latter does not organise their peers into an overlay and as a consequence, finding peers is slower compared to structured peer-to-peer systems. Yet, some systems use unstructured peer-to-peer systems [7, 13, 32].

Systems that are focused on performance use overlays to structure their peers. Such overlays are deliberately created to allow for efficient and fast lookup of peers and their data. There are different ideas for creating such topologies.

Chord [39] organises peers into a ring, while Kademlia [25] uses a binary tree to achieve the same goal. Both solutions provide logarithmic lookup time. The common thread between most peer-to-peer overlays is the fact that they provide functionality similar to distributed hash tables which are discussed in Section 2.2.

1.2 Distributed hash tables

Most Distributed hash tables, hereafter referred to as DHTs, e.g. Chord and Kademlia are used to store small objects, usually under 1 MB. However, we wish to enable DHTs to store large files. DHTs have a simple put, get, remove interface. A *put(value)* request sent to DHTs are routed by hashing the value and routing the request to a peer with identifier equal or close to the hashed value. This peer can be designated as the *coordinator* for that key. The coordinator is usually responsible for replicating the value across the DHT and placing them according to an application-specific policy. This procedure can be very heavy on the coordinator if peers have low bandwidth and the number of replicas are relatively large. This may be the reason why DHTs are mainly used to store relatively small files.

1.2.1 Motivation

Enabling DHTs to store large files is important for many multimedia applications. However, the coordinator has to propagate large files to peers across the DHT. This procedure is expensive, in terms of bandwidth, and creates a hotspot, which leads to an imbalanced DHT. Hotspots occur when a segment of the network is more active than the rest. The coordinator also has to create replicas under churn. Churn is defined as an event occurring to a large set of peers. In this thesis, churn can be due to peers leaving or changing interest. Aside from peers leaving the system, new replicas must be created as peers interest shift drastically and a file becomes very popular. A third scenario exists if the DHT enables mutable items to be stored. In such cases an *update* interface is added to introduce updates. Large updates must also be propagated to keep replicas consistent. In DHTs where updates are large and relatively frequent, the coordinator can easily become a hotspot.

Our motivation is to enable to DHTs propagate large files in short time, while keeping overhead low and keeping the DHT load-balanced. In Section 4.3 on page 20, we will look at the different available techniques and their flaws in such cases.

1.3 Contribution

Our contribution in this thesis is a propagation scheme for DHTs, where factors such as peers' bandwidth and files size are taken into consideration. This is done by dividing large files into pieces and enabling interested peers to collaboratively exchange pieces among each other. We focus on highly available replicas as they are better suited for peer-to-peer systems. Generally, consistency types can be grouped into two, strong and eventual consistency. Our protocol is evaluated using eventual consistency, however a strongly consistent system e.g. Paxos may also benefit from this protocol. We describe different types of consistency in Section 4.2. Security of the protocol is outside the scope of this thesis. We assume peers are cooperative and honest with no malicious intent. However we do not alter any of the fundamental concepts in DHTs and the security techniques used in standard DHTs still can be deployed on top of our solution. We further assume that there is at least one peer with a full copy of the file that is being propagated is accessible and is participating in the propagation.

1.4 Related work

The research done in this field is mostly focused on mutable DHTs[4, 5, 26] or a DHT with transactional consistency [26]. To our knowledge there is no good solution for propagating data in DHTs. In cases where values are small, this is an insignificant issue to research. However as the size of files stored in DHTs grows, the load-balancing property of DHT suffers. The other factor that must be considered is the bandwidth available to peers. If a coordinator has a low-bandwidth connection, propagation will take a long time.

1.5 Methodology

As we wish to evaluate our protocol in a relatively large peer-to-peer system with numerous files, it is time consuming to have long-period experiments run in real-time. In addition, there are many different experiments that need to be run. To save time, we have implemented a simulator to simulate the real-world events. We have used a subset of Wikipedia dataset to evaluate our work. We will evaluate our protocol while focusing on two metrics, propagation time and overhead introduced by our protocol.

Chapter 2

Peer-to-peer systems

In this chapter we will generally describe some of the advantages of using peer-to-peer systems. We will describe two types of peer-to-peer systems with focus on DHTs discussed in Section 2.2 on page 10.

Peer-to-peer systems are known by the fact that each peer in the system provides the same functionality as others. Peer-to-peer systems usually offer services such as storage, routing, processing power, etc. The simplicity of peer-to-peer architecture makes it easy for developers to build more complex applications on top of structured peer-to-peer systems e.g. storage services, multimedia streaming, online video games, etc.

An advantage of peer-to-peer systems is that they do not require any servers or abnormally large infrastructures to operate. Peer-to-peer systems offer great scalability, because workloads are distributed among the peers in the system. Another great advantage is the load-balancing offered by peer-to-peer systems. Many proposed peer-to-peer overlays such as [25, 31, 34, 39, 42] provide excellent load-balancing. The overall robustness of such systems is increased because there is no single point of failure. In addition, failures and churn are anticipated and handled. Computer crashes or Byzantine failures [24] can no longer bring down, maybe not even slow down, entire systems. Byzantine failures are lower class of failure where systems can receive or process requests incorrectly which leads to incorrect output. As we are working on a peer-to-peer environment, peers can join or leave the system voluntarily. Events such as peers joining or leaving the system at the same time, or in a short time, must be expected and handled accordingly. Such events are also known as ”churn” in peer-to-peer networks.

Generally peer-to-peer systems can be grouped in two categories:

Structured peer-to-peer: There are many overlays to structure peers. Chord [39], Kademlia [25] and Pastry [34] are among the most popular overlays. All of them provide DHT functionality. Using the simple put/get interface provided by DHTs, it is easy for developers to build other services on top of DHTs, such as file systems [10, 27, 38], peer-to-peer search engines such as MINERVA [3] or caching [17].

Unstructured peer-to-peer: In these systems, nodes have no prior knowl-

edge of network overlay as they join. There are also many systems in this category such as file sharing networks. For instance, Gnutella [32], Kazaa[19], eMule[13], Freenet [7] can share files without structuring peers. Music streaming service Spotify [20] and VOIP service Skype[36] also avoid structuring their peers.

Although, unstructured peer-to-peer systems are outside the scope of this thesis, we will briefly describe them for the readers to gain a better understanding.

There are many known schemes to locate data in unstructured peer-to-peer systems, such as *flooding*, *random walks* and *expanding ring*. Flooding is the simplest way to locate content as queries are sent to all neighbours. Random walk is a technique where a "walker" is initiated to randomly walk through the network and (hopefully) locate the interesting data. Expanding ring is very similar to flooding, but queries are limited with hops-to-live fields which are decreased by one, for each hop. The query is eventually dropped if the data was not located within the limit.

An advantage of using unstructured peer-to-peer systems, is that they are more resilient to node failures than structured peer-to-peer systems. Because of their lack of topology, failures do not affect the topology. On the other hand, many of the techniques used in such systems do not guarantee locating data. In addition, they create a lot of overhead, due to the large amount of messages sent. Also, the latency of locating data is much higher compared to structured peer-to-peer systems. All these techniques can create a lot of overhead and are time consuming. However, unstructured peer-to-peer systems may also benefit from our protocol by reducing the propagation latency.

In the next sections we will discuss structured peer-to-peer systems and their advantages and disadvantages. We will also talk about how structured overlays are better suited for our environment. We will discuss how structured peer-to-peer systems provide DHT functionality and discuss their advantages and what might be lacking in such DHTs.

2.1 Structured peer-to-peer

In such overlays, peers are organised and values are **not** placed at random peers. Properties such as *peer-ID* are used to determine which values peers host. A simple *get* and *put* interface is used to store and retrieve values from DHTs. Also a *remove* procedure can be used to remove keys from the DHT.

Most peer-to-peer systems have a strongly structured overlays. An advantage of organising peers is to avoid broadcasting requests and utilise peer bandwidth efficiently. In addition, by organising peers data can be looked up quickly. Chord and Kademlia are well known DHTs that offer logarithmic look up time.

Although our propagation protocol is orthogonal to the DHT implementation, we will describe Chord to illustrate how peers can organise

themselves and offer a DHT-functionality. We will keep using Chord to illustrate components and scenarios in the rest of the thesis.

2.1.1 Chord

Chord [39] is simple to understand and very efficient and resilient to failures. It is also robust again churn, which was briefly mentioned in earlier in Section 1.2 on page 4.

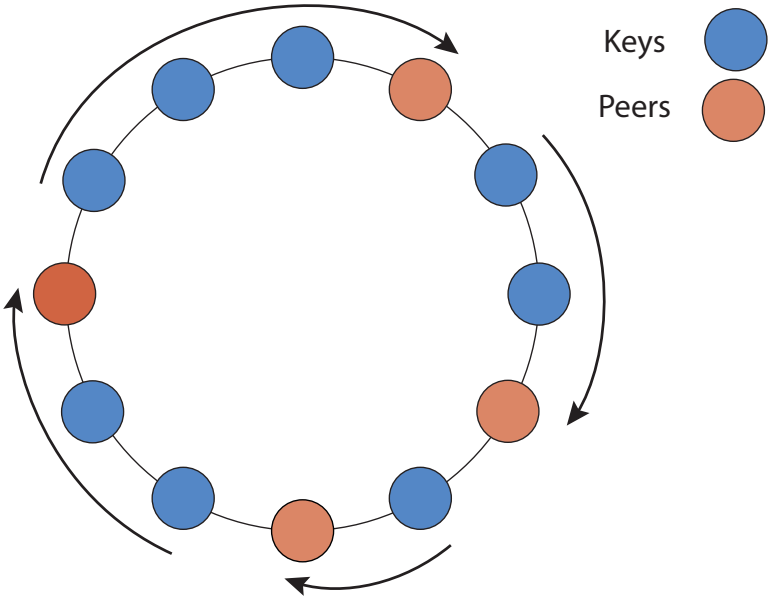

Figure 2.1: A possible configuration for a Chord ring.

Chord is a look up service for peer-to-peer systems with lookup time proportional to $\mathcal{O}\log(n)$, where n is the number of nodes connected in the system. In Chord, peer identifiers are a hash of their contact information. For instance, IP address, port and transport protocol e.g. TCP or UDP might be considered a contact information. The peer identifier is used when storing data.

In Chord, every peer has a pointer and contact information for its successor. A successor can be found by comparing peer identifiers. The peer with the highest identifier has a pointer to the peer with the lowest identifier. Thus, a logical ring is created. A clockwise walk across the ring, will result in a list of peer with increasing identifiers. However, this solution does not scale well. To resolve this issue each peer also maintains a *finger table* which points to peers with distance 2^{i-1} across the ring. Obviously, i is in range of $<0, \log(n)>$. The *finger table* is used as a short-cut to peers across the ring. Using the finger table, peers can route messages efficiently and in logarithmic time.

Generally, a *put*(value)-procedure hashes the value and creates a key. If a peer with the same *peerID* as the key is online, the <key,value>-pair is stored at that peer. This way, the *get*(key)-procedure only needs to locate the peer with identifier similar to key. However, if there is no peers with

the same identifier as the key, that <key,value>-pair will be stored at the closest peer. As can be seen in Figure 2.1, there are many more keys than peers. <key,value>-pairs that do not map to a specific peer, will be stored at the closest peer when walking in a clockwise direction.

When a peer receives a *get*-request, it first checks if the matching value is available locally. If that is not the case, then the finger table is used to re-direct the message to the closest node in the table. This operation is repeated until the matching value is found and returned to requester. Obviously, if the key requested does not exist, an error message will be returned. Some DHTs can create temporary replicas along the routing path to minimize the amount of hops needed for a response. A full description of the Chord protocol and it's evaluation see [39].

2.2 DHT

Advantages of using distributed hash tables is in their simple interfaces and the powerful properties they provide such as scalability, availability, load-balancing etc. There are also some disadvantages when using DHTs. DHTs provide *write-once* and *read-only* semantics. In other words, DHTs only store immutable data. An immutable data is an object that can not be changed. Having mutability in data is an important issue in DHTs. Another type of DHTs are the kind that store mutable data. In terms of interface, the difference between mutable and immutable DHTs are minimal. Both of them offer a *put*(key, value) function to store <key,value>-pairs and *get(key)* for locating and retrieving values. Both of them offer a *remove*-procedure to remove the <*key,value*>-pair. Adding *update*(key, value) procedure to DHTs is an important issue that has been researched in the recent years. Implementing an update procedure raises the issue of how updates should be propagated. Our propagation protocol can also be implemented on top of mutable DHTs to propagate both large files and large updates. The new propagation scheme helps to keep the DHT load-balanced and reduces the latency.

2.3 Summary

In this chapter we have talked about peer-to-peer systems in general and briefly described two different types of such systems. We have illustrated and described how Chord can organise peers and offer a DHT functionality. In Section 2.2, we discussed DHTs in general, what they are missing and how an efficient propagation scheme can improve both types of DHTs, mutable and immutable.

Chapter 3

Publish Subscribe systems

In this chapter we briefly describe Publish-subscribe systems, hereafter referred to as pub-subs. We will introduce SpiderCast [6] as an overlay originally suggested by authors to be used in pub-sub systems. However, SpiderCast has many applications, some of them are described in Section 3.2.1. In Section 3.3 we will talk about how we are adapting this overlay to our needs and the benefits it offers us.

Publish subscribe systems are considered event-based architecture. In such systems there are two groups, which we will briefly describe in the following sections.

3.1 Publish subscribe systems

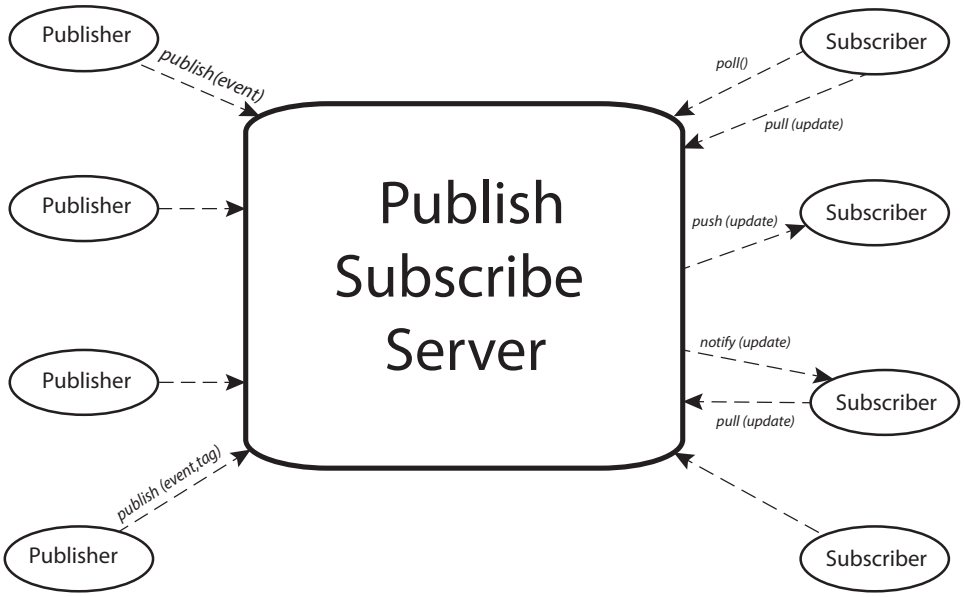

Figure 3.1: An abstract view of publish subscribe systems and different interfaces that can be implemented.

3.1.1 Publishers

Publishers are the information providers. The example most widely used to illustrate a publisher in pub-sub systems are news media. Many news-related articles are created each day and some of are them updated afterwards. It is important for such publishers to notify users about these articles as soon as possible. Publishers can use a function similar to *publish(event)*, to upload the event to the pub-sub systems. In some pub-subs, publishers may have the ability to use *tags* describing the content of the article. This is can be done by a function similar to *publish(event, tag)*. Using such tags, subscribers can define an interest domain and receive updates or news that fall within that domain. Another possibility is to create tags for the article inside the pub-sub server. This requires no effort on publishers to tag documents. However it can create a lot of computation overhead on the servers.

3.1.2 Subscribers

Subscribers are the group receiving the information and updates. They show interest by sending a filter. Filters can be defined in different ways. For instance a channel-based pub-sub is a system where subscribers are interested in every event from a specific publisher. There are also topic-based pub-subs where subscribers provide the system with tag(s) and receive all events, articles, etc. with those tags.

There are different ways subscribers can be notified. Some of them are illustrated in Figure 3.1. The push method has the lowest delay, but also has some disadvantages that we will discuss in Section 4.3.1. In the pull scheme, subscribers must *poll()* the server to check if newer content is available. If there is a new article or an update of one, a *pull*-request is sent. In addition, anti-entropy protocols can be used to exchange information. Although originally introduced as a multicast communication scheme, anti-entropy protocol has many applications. We will describe each of these protocols in detail in Section 4.3 on page 20.

3.2 SpiderCast

SpiderCast [6] is a protocol used to create an overlay in topic-based publish subscribe systems, which is scalable and interest-aware. Although publish subscribe systems are used to explain SpiderCast, it has many more application as we will see in Section 3.2.1. SpiderCast is a fully decentralized overlays and requires no central server. As a result, SpiderCast handles failures very well. Peers using this overlay do not need complete view of the system, only partial view. This enables SpiderCast to scale well as the system grows. This overlay creates a sub-overlay for each topic, ensuring that subscribers who are interested in the same topic are connected.

The idea behind this overlay is that random graphs can be connected if each node had only three or more greedy links. In other words, if a node

has 3 (or more) greedy links, it is highly probable that the entire network is connected. Greedy links are chosen greedily to maximise the use of a link.

In SpiderCast each node continuously adds links to other peers until there are K different neighbours interested in the same topic. When we have links to K-neighbours interested in the same topic as us, we say that topic is K-covered. There are two different neighbour selection methods, greedy and random. If K-neighbours were selected using the greedy selection method, we can minimise the number of links for all topics. Using these links, topics might partition. When a topic partition occurs, there are two sets of peers unaware of the fact that both are interested in the same topic. By choosing K-neighbours based on random selection method, we get a robust sub-graph for each item. However, the topics we cover for each link would decrease, meaning that we do not cover as many topics as we wish when neighbours are selected randomly. It must be noted that random neighbour selection considers only peers with at least one topic in common.

SpiderCast can provide high probability of topic connectivity. In context of pub-subs, topic-connectivity means that nodes interested in the same topic are connected either directly or through neighbours. Evaluation of SpiderCast shows that the average number of neighbour increases in a logarithmic manner as topics grow. In addition the average number of neighbours per peer slowly decreases as number of grow. The average number of neighbours per node is 45% smaller than other overlays used to support decentralized pub-sub systems.

For more details about how the overlay works, see "SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication" [6].

3.2.1 SpiderCast applications

SpiderCast can be used in group membership services and as a multicast overlay. Members can form an overlay using SpiderCast, and benefit from the churn-resilience and efficient data dissemination. This overlay has also been proposed as an overlay in peer to peer social networks. In such systems, a sub-overlay can be created for each peer. Peers can show their interest to other peers by becoming a member of their sub-overlay. This way friends can visit profiles and receive updates pictures, etc. from that peer using the overlay. Applying SpiderCast in peer-to-peer online social networks was proposed in [29].

SpiderCast has also recently been implemented in collaborative Wikipedia hosting [41]. In that system SpiderCast was used to provide a sub-overlay for each Wikipedia file. An Wikipedia entry can include pictures, videos, text or other types of documents in the entry. Since there are millions of Wikipedia articles, a scalable overlay was required. The proposed architecture re-defined topics as Wikipedia entries. Peers interested in an entry, is a replica for that item. User requests can be routed to a replica. The architecture proposed in [41] considerably reduces the required traffic to serve all requests originated by Wikipedia users.

3.3 Our adaptation of SpiderCast

In peer-to-peer systems bandwidth is sparse, and we must gain as many benefits as possible from each link we create and maintain in our DHT. The original paper states that "the average node degree in SpiderCast is at least 45% smaller than in other overlays"[1]. This is a huge benefit in peer-to-peer systems, because bandwidth is sparse.

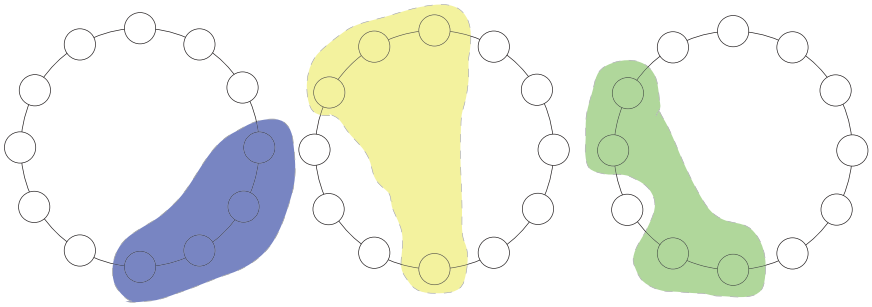

Figure 3.2: Example of three possible sub-overlays created by SpiderCast

In our DHT implementation, interest is defined as values stored at peers. In other words, if a peer hosts a key, it is said that this peer is interested in that key. By using this definition and using SpiderCast, we can create a sub-overlay for all peers who host a specific key. In other words, replicas for a specific item will be connected to each other, either directly, or through other replicas that have the same interest. Figure 3.2 shows three possible overlays created by SpiderCast over a DHT. The ring in the figure illustrates a Chord overlay and the shapes on top of the ring represent different kind of overlays SpiderCast can create to connect replicas with common interest(s). We emphasise that our solution is not dependent on any specific DHTs and the chord ring in Figure 3.2 is only illustrative.

The propagation protocols describe in Section 4.3 can be used to disseminate information using the SpiderCast overlays. However problems arise when propagating large files by using such protocols. We will discuss propagation protocols in Section 4.3 on page 20.

A difference between the original SpiderCast an our approach is how we define coverage. In the original paper, a neighbour provides coverage for an item if it is interested in it. However, we have a more strict definition of coverage. In our approach a replica provides cover if it hosts the value *and* has pieces that might be interesting to us.

In the original SpiderCast the maximum number of neighbours were defined as $k * h$, were k is the coverage we wish to have, and h is the number of values. Because there can be thousands of values stored in the DHT, this approach is not practical. In our solution the maximum number of neighbours is configurable according to some pre-set parameter that we can configure in the simulator. We describe some of the parameters in Section 9.5 on page 70.

[1]Node degree is a term used to indicate the number of connections for a node

Another difference between our approach and the original SpiderCast, is when the overlay stops adding links. SpiderCast disconnects from redundant links if every topic is K-covered. The disconnection procedure is activated when the number of neighbours exceed the maximum. In our approach the disconnection procedures is triggered when (i) K-coverage for every topic is achieved and (ii) if the connection procedure has been executed multiple times and in event K-coverage was not achieved.

3.4 Summary

In this chapter we have briefly described pub-sub systems and the interfaces available on this architecture. We have also introduced SpiderCast that was originally suggested by authors as a pub-sub overlay and described how it can be applied to peer-to-peer systems as an overlay for online social networks. We also described how we will use this overlays to keep replicas interested in the same files connected. Our approach was had re-defined the concept of coverage compared to the original SpiderCast.

Chapter 4

Replication

Replication is a basic technique of having multiple copies of the same file on different peers. This improves availability and performance in a system. Systems built on dynamic environment, mass join and leave are likely to happen. When a replica dies another replica can process requests and provide the necessary data, thus increasing data availability. Another advantage of using replication is the performance increase. Replicas can share the workload for an item to decrease the response time, improve load-balancing and make the system scale better.

The main challenge in this thesis is to propagate information to replicas that are involved in hosting that item. As a consequence, replication techniques are a central part of this thesis. In this chapter we will briefly explain different replication techniques and semantics.

Replication is widely used in distributed file systems [11, 21, 22, 27]. This is mainly because peers can join and leave the system freely at any time. Each peer will host <*key*,*value*>-pair(s). When peers leave the system, the data hosted by the leaving peers should not be lost, hence there is a need to replicate data to avoid such loss.

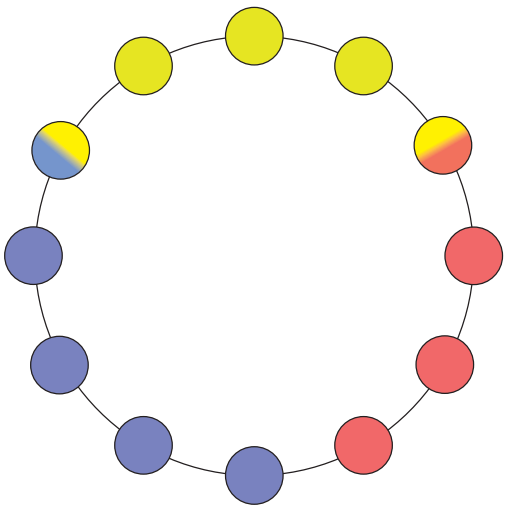

Figure 4.1: An examples of possible replicas in a DHT

Replication policy is a set of requirement set by system designer that have to be met by a system replicating data. Some examples of requirements can be the number of replicas that must be accessible at any time in the system. This is also called replication factor. The policy can also set a limit on system latency for serving user requests. Another possible requirement is *where* replicas are placed in the event of churn.

Replication improves performance in two ways. It helps scale the system as more users join the system. Replication helps systems scale by placing replicas close to users. As the number of users in a system scales, the replicas help by responding to user requests. Replicas also help the system scale in a geographical manner. Replication also improve load balancing. By enabling web servers to re-direct traffic to different replicas, a system can balance the workload among available resources.

On the other hand, data replication is not without a price. If a replica of a file is created, it needs to be kept *consistent*. There are different types of semantics used to achieve consistency and we will briefly go through some of them.

4.1 Coordinators

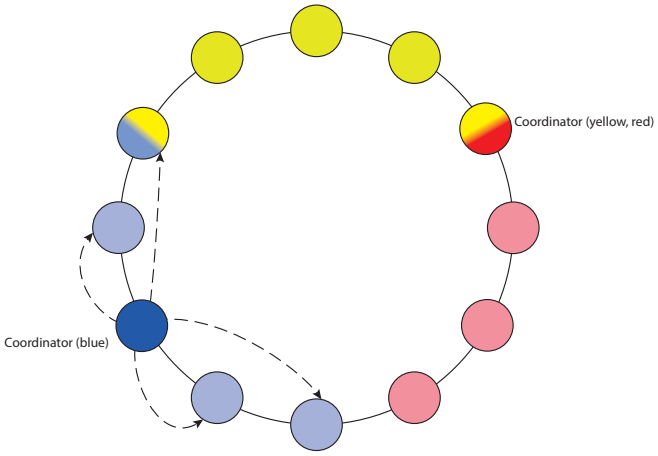

Figure 4.2: How coordinators can propagate files to peers in DHTs

Figure 4.2 illustrates a Chord ring where the *coordinator* is responsible for propagating fa files to peers in order to create new replicas. This is done according to the replication policy implemented in the system. Due to churn and to maintaining data-availability new replicas of files must be created to avoid data loss. In other cases where a song or a video goes viral and becomes extremely popular in a relatively short period, files have to be replicated to balance the workload among peers in the system. Possibly coordinator have to propagate the update to replicas if such events are allowed in the DHT. Although most DHT do not support storing mutable data, there have been some proposals [4, 5, 27, 38]. Our work can benefit the *coordinators* by reducing the propagation load and latency.

4.2 Consistency types

4.2.1 Strong consistency

Strong consistency is a type of consistency where all users will see the same copy of a file independent from the source that served the file. This gives users the illusion that there is only one copy of a file. This consistency type is called one-copy consistency. Such consistency types can be achieved simply by making replicas unavailable until they can prove that they have the most recent version. When discussing strong consistency, there are four properties that must be introduced. ACID-properties are four important properties that were first introduced by Jim Gray in [15] to indicate what properties an ideal transaction must have. They are mostly used in database systems, where the consistency is very desirable. ACID-properties include:

Atomicity: is also known as ”all or nothing”-property. This property implies that transactions or updates are either successfully finished or rolled back in case of failure. In other words, either the *entire* transaction is completed or *nothing* is changed.

Consistency: is a property that ensures the validity of the database system. Given a set of constraints for the system, consistency property ensure that the system moves from a valid state to another valid system.

Isolation: can be thought of as a constraint for concurrency control. A failed transaction will not be visible other transactions.

Durability: is the property of committed transactions will remain so. In case of failures such as computer crashes or network issues, the committed transactions will not disappear and endure the failures.

Financial systems, such as banks or stock exchanges, benefit from such properties because there is an emphasis on transactions and database systems always being in a valid state. In such systems, the need for consistency outweighs the need for availability. This results in a certain amount of unavailability of date during update processing. Although updates are small in financial systems, a problem can occur when processing updates large in size.

Although there have been some proposed solutions to achieve strong consistency in DHTs, we will not use such semantic to provide consistency. Providing strong consistency in peer-to-peer systems can decrease availability, especially in environments where replicas of files are created and removed dynamically. Factors such as file size, peer bandwidth and dynamicity of environment can decrease system performance. Our aim is to take such factors into account and we wish to have low overhead. As strong consistency introduces relatively large we will not provide strong consistency nor ACID-property for our updates. We will focus on availability in out mutable DHT.

4.2.2 Eventual consistency

In eventual consistency the only guarantee provided, is that replicas will eventually converge to the same state, if a new updates has not occurred. In systems where such consistency is deployed, it is possible for users to read different values from different replicas. While in some systems that is unacceptable, such as financial institutions or banks, many websites such as Google, Wikipedia, Amazon might prefer availability over data strong consistency. Eventual consistency has been implemented in Bayou. The architecture was meant for "data sharing among mobile users" [12]. Eventual consistency can also be used in systems such as, Cassandra [22], which is a highly available storage system. Cassandra was originally proposed as a storage system for Facebook. However, it was released as an open-source project in 2008. According to Apache website [2] Cassandra is used by Twitter, Netflix, eBay and many more.

In this thesis, we will focus our efforts on *eventual consistency*. The reason is simply because of factors such as file size and peer bandwidth and dynamicity of environment both in term of popularity of items and churn. Popular items have to be replicated quickly in order to serve flash crowds. These factors combined with the communication overhead introduced by the quorum approach or consensus protocols pushes us toward a solution that with less protocol overhead while using the bandwidth efficiently.

4.3 Consistency protocol

Different propagation protocols have been proposed through time. In the following sections we will briefly describe different concepts for propagating information to replicas. Note that these techniques can be used to provide any type of consistency, depending on the system they are deployed on.

4.3.1 Push protocol

Push-based consistency protocols are also known as server-based protocols. This protocol can be used by servers to update their replicas. An entity (usually a server) has to receive updates, order them and push the data to replicas. This protocol is used when a high degree of consistency is required. In push protocol replicas of a file have to be created and maintained by an entity. A server needs to have a list of all replicas for every item available on the server (we assume that every item is replicated). As a result, a list of all the replicas and all the items must be maintained by the server.

As peer-to-peer systems are server-less and completely decentralised. However, the push protocol can be deployed in peer-to-peer systems by doing some modifications. We can designate a peer to be the coordinator when propagating information in DHTs. Having only one sender when propagating in DHTs required the sender to offer a lot of bandwidth to do so. This creates a load-balancing problem, resulting in hotspots in the

network. Hotspots occur when a segment of the network is more active than the rest. Although this solution does not require a server, it has the same problems. It does not scale well as the file sizes or number of files increase.

4.3.2 Pull protocol

Pull protocols are also known as client-based protocols. In this protocol replicas periodically poll a server to see if an update exists. If so, replicas ask the server for content of the new update. Although this can create some overhead, it offers replicas the chance to receive updates when it is best suited for them, unlike push-based protocols.

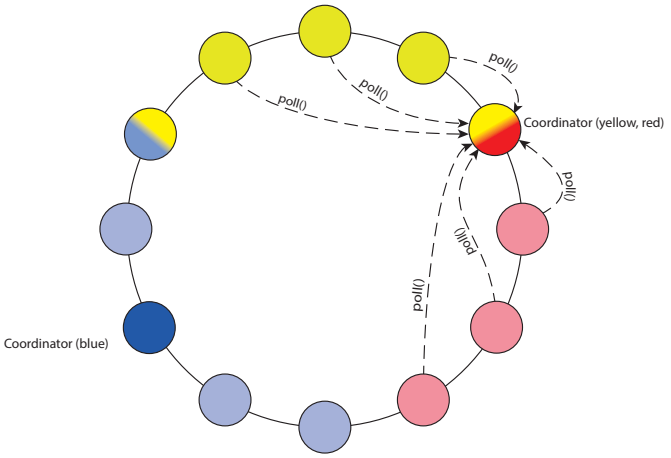

Figure 4.3: How updates can be pulled from primary replicas

A disadvantage of such protocols is that they provide lower degree of consistency than pushing. This means that the delay between the time an update is introduced, and replicas having the content is relatively large. As this delay increases, so decreases the degree of consistency. Such protocols can be optimised by increasing the frequency of polling from the server. Although this solution provides a stronger consistency, by decreasing the delay we mentioned earlier, it can introduce a lot of overhead to the system. Every time the server is polled when there is no new update(s), the message is considered as communication overhead.

In peer-to-peer systems where peers might host multiple files, peers with the new update(s) will be required to meet the huge traffic demands created by requests from the other replicas. This solution also does not fit the peer-to-peer architecture, as it creates a hotspot in the network and does not make efficient use of the sparse bandwidth by making a lot of requests.

4.3.3 Anti-entropy

Anti-entropy is an information dissemination protocol that is mostly used in epidemic information dissemination [14], where no overlay is needed to

disseminate updates. In anti-entropy protocol a node, a, randomly chooses a node b and exchanges updates with b. The exchange starts with nodes pushing small messages to each other. The content of messages pushed, is the current state of its sender. After knowing of each others' state, nodes can pull the data they require.

In our solution a push-pull based approach is implemented. In this approach, peers push their state only to interested peers. Receivers of push message can decide to choose the time and the source from which they request the content from. Using the push-pull approach, we can benefit from notifying peers about events quickly while avoiding the overhead created by using periodically sending poll-messages in pull protocols. In our solution peers can choose which peers to pull from, if there are multiple peers whom have new information. We will provide a high-level description of this protocol in Section 7.3 on page 43.

4.4 Summary

In this chapter we have introduced the concept of replication, different consistency types, protocols. We have described the reconciliation process, which we will use with a combination of push-pull based protocol that provides low latency for update notification. It also allows peers to choose the source as well as the time for sending a request to get the new update.

Part II

The project

Chapter 5

Problem definition

In this chapter we will describe the environment we are working on and give an overview of functional and non-functional requirements in the upcoming sections. But first, we will define the problem we are solving.

As mentioned earlier in Chapter 4 replication is widely used in peer-to-peer systems. As peers leave the system, replicas of a file might disappear. DHTs must consider this problem and create new replicas of files as peers leave. However a problem arises when the file stored is relatively large. The *coordinator* that was previously described in Section 2.2, is usually responsible for propagating a copy of large file to new peer(s) to create a new replica(s). This requires the *coordinator* to offer a lot of it's bandwidth. Another issue is low-bandwidth peers that do not receive data fast and delay the *coordinator* and in worst cases the entire DHT. Our aim is to propose a protocol to improves propagation of relatively large file in DHTs. This protocol can also be used to propagate quickly in case of flash-crowds or viral videos. As an audio or video file gets more popular, users will request that particular file more frequently. In cases where videos go viral, this increase is tremendous and sudden. A peer-to-peer system storing and handling multimedia files has to have many copies in order to serve all users. Even more copies might have to be accessible if the requirements for quality of service are strict.

Another benefit of quickly propagating relatively large files in DHT is in cases where DHT host mutable data. Mutable data, are data that can be modified. Such DHTs must have an update-interface and the necessary consistency management protocol in order to store mutable values. All mutable DHTs have to propagate the updates among the replicas that host the recently-updated value. In cases where updates are relatively large, our protocol can be used as a propagation mechanism to propagate updates received from users. We emphasise that our work is orthogonal to consistency types and serialisation techniques used in mutable DHTs and our protocol can be used in both mutable and immutable DHTs.

The rest of the chapter is as follows, in Section 5.1 we will describe what kind of components are required as basis for our protocol. Some of the description may be vague, but that is only because the environments are simple and standard. In Section 5.2 we will discuss some functional

and non-functional requirements that need to be satisfied for a propagation protocol.

5.1 Environment description

In this section we will describe some assumptions about the underlying architecture, interface and some components that fall outside the scope of this thesis.

5.1.1 DHT

Our solution assumes a DHT implementation. However it is orthogonal to the implementation itself, as no implementation-specific functionalities are used. By avoiding the modification of specific implementations, our work is cross-operable between any DHT. Basically, we assume any DHT with a standard *put*, *get* and *remove* interface. DHT implementations can have any underlying structures e.g. Chord's ring structure or Kademlia's binary tree. In the next section we will look at the standard interface provided by DHTs and look at some standard properties they offer in Section 5.1.1.

Standard interface

By building on top of immutable DHTs, we can extend the interface by providing an *update* interface while preserving all other interfaces. In the next sections we will briefly describe what we require from these functions.

The *put* interface is called when a DHT client wants to insert a value in the DHT. The value is hashed and the <*key,value*>-pair is stored at a peer with a peer-identifier *close* that is to key. The same closeness relation is later used to lookup that value. DHTs replication policy might have to create replicas of the <key,value>-pair immediately after insertion, although that depends on the implementation.

The *get* procedure is called to retrieve values previously stored. This is done by locating the peer storing the key, which is done by exploiting the same relationship between peers identifiers and keys stored in it. Given a lookup request for a specific key, peers will route the request to the closest neighbour in their routing tables. This routing procedure is executed for each peer in the path until the <key,value>-pair is located and returned to the user.

The *remove* interface will remove the <*key,value*>-pair from the DHT. There might be different ways this to accomplish this. One way is to assign every <*key,value*>-pair an expiration date. When reached, the *coordinator*-replica will not renew the <*key,value*>-pair, thus it will be eventually removed. Another approach for the *coordinator* to actively send a *remove*-message for all replicas for that item.

Standard properties

DHTs have different implementations, but we assume that the underlying DHT has a set of properties that remain constant. These properties are abstract and may be implemented in many ways. They are considered as standard for many DHTs implemented today. They are:

Load-balancing: DHTs enable every peer to do any of the operations we previously mentioned. As a result DHTs support multiple writers and readers. By using replication it is possible to provide sufficient load-balancing for popular files.

Churn-resilience: DHTs are very robust to churn. This is because of its fully distributed architecture and balancing the load evenly among peers in the DHT.

Scalability: By using consistent hashing[18] DHTs are incrementally scalable while maintaining the same properties and offering logarithmic lookup time.

Availability: Although peers are always available, data can be lost if there is no replication scheme used. Data availability is mostly achieved by using replication techniques. However using *strong consistency* may limit the availability of data, since data is made unavailable during updates.

5.1.2 Replication policy

An essential component relevant in DHTs is replication policy. This policy is a set of requirements that are defined by the system designers. It includes a policy and mechanisms that help enforce that policy on the DHT. It needs to monitor the responsiveness, load-balancing and data-availability of DHT and the files in it. In event of churn or flash-crowds, some procedures must be executed to keep the system within the defined policy.

As peers leave the DHT or peers crash due to churn, replicas will be inaccessible. Files must be propagated in order to create new replicas and maintain the replication factor defined by the replication policy. In cases where files become popular, a similar propagation has to take place to create replicas and respond to user requests quicker and improve the quality of service of the DHT. In addition, a replication policy can remove replicas as files become less popular. This helps in reducing the replication redundancy in the system. Another responsibility is to re-place replicas to improve the quality of service. For instance replication policy can lower the response time by placing replicas closer to users.

These mechanisms and the policy are both outside the scope of this thesis. However, we assume that such policy exists and provides sufficient load-balancing to keep the DHT load-balanced as peers serve user-requests. We also assume that the replication policy uses a dynamic replication factor and can create replicas of files in runtime.

5.2 Requirements

In this section we will talk about some of the functional and non-functional requirements that should be met in order to provide a good solution for propagation within the DHT. We will also define the scope of the thesis.

5.2.1 Functional requirements

Our functional requirements are not complicated. We require that the large data being propagated in the DHT is received by all the interested peers. In our case, a peer interested in a file is considered a replica. In cases where new replicas of files need to be created, we need to propagate the data to interested peers.

Generally, we wish that updates are propagated only through replicas that are interested in the update. In other words, if a replica r, has no interest in file f, then it should not participate in the propagation of that file.

We summarise this section with two simple requirements. We require that the file will be eventually delivered to the new replica if it does not leave the system. We also require for our protocol to only involve peers who are interested in that file and no other peers. We have compiled a full list of requirements in Table 5.1.

5.2.2 Non-functional requirements

There are many non-functional requirements for our solution. The essential requirement is to not sacrifice or degrade any of the properties offered by a standard DHT. In other words, we wish to preserve all the properties offered by immutable DHTs, especially during propagation of big files.

Churn-resilience

As we previously mentioned, DHTs offer excellent resilience to churn. We do not wish to degrade the robustness and failure tolerance of the overall system. Our work does not affect the resilience of DHTs as it does not alter any failure detection nor recovery mechanisms.

High data-availability

Generally, this is achieved by using replication. However, using *strong consistency* can limit availability in case of updates. In addition, creating replicas dynamically and during runtime is important to adapt to dynamic environments. But, propagating the files to peers can take a long time when peers have low-bandwidth connections. Our work will improve data-availability by reducing the propagation time needed to create or update replicas.

Load-Balancing

DHTs have an inbuilt load-balancing technique by distributing files evenly among peers. To balance the load for frequently read files, replication techniques can be used. New replicas must be created to serve popular files. However a load-balancing problem arises when data has to be propagated to the new replica. The *coordinator* is responsible to propagate the entire file to peers. The network load of propagating large files will create a hotspot in the DHT, especially when the environment is dynamic. In such environments, a number of replicas might be needed quickly to serve a flash-crowd or viral videos. The worst case scenario occurs when the *coordinator* must solely propagate the file to several peers at the same time. In Section 6.2.3 we will see how our work improves the load-balancing when propagating relatively large data in DHTs.

Scalability

DHTs are known to scale incrementally well. However, that might limit data-availability. This is a typical trade-off between scalability and availability. The load-balancing issue discussed earlier could lead to scalability problems. As the number of files in the DHT grow and more files become popular, many hotspots can be created and severely decrease the scalability of the DHT.

Efficient bandwidth utilisation

In DHTs, some peers might have low-bandwidth connections. It is important that these peers do not slow down the propagation by forcing the coordinator replica to transmit the data at a lower rate that suits the newly instantiated replica. On the other hand, the exact opposite situation can occur where the coordinator has a low-bandwidth connection and can not propagate to several replicas quickly. Figure 5.1 illustrates the two cases where propagation can be delayed because of slow senders or receivers. We will focus on utilising peers' bandwidth efficiently, as different peers have different bandwidth constraints.

Low overhead

Low overhead can be achieved by using protocols and algorithms not requiring frequent, large or many control messages. We will analyse this property closer in Section 6.2.6 on page 39.

5.2.3 Scope of problem

Making a DHT secure is generally a challenging task. Because they are fully decentralised, it is difficult to protect DHTs from malicious peers. However, we believe that security challenges in DHTs are orthogonal to our work. For this reason we do not consider the security of our solution. In other words, we assume that peers in our systems are honest, and

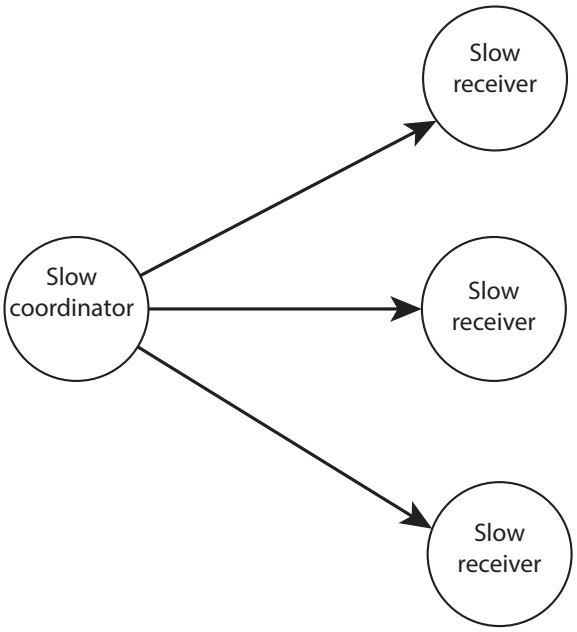

Figure 5.1: An illustration of how slow sender or slow receivers can effect the system.

willing to allocate their resources accordingly. For more details about vulnerabilities of DHT and security techniques, we recommend "A survey of DHT security techniques" [40].

The replication policy required in section 5.1.2 is also outside the scope of this thesis as it is very application-specific and complex. However, we assume that propagation of data is needed as our focus is on propagating relatively large data in DHTs. Propagation within the DHT can occur when replicas are updated or instantiated at peers that do not have the file. We wish to propose a quicker propagation protocol where peer bandwidth in taken into account, the overhead is kept low and propagating load is balanced among replicas. Finally Table 5.1 shows some of the requirement we have defined and the priority we have assigned them.

5.3 Summary

In this chapter we have gone through some properties and functions we require from the underlying DHT. Although these requirements might seem difficult to meet, most DHTs e.g. Chord and Kademlia already offer all of them. We emphasise that these requirements are considered as standard in most DHTs. In addition we have compiled a table of required properties.

Requirements	Priority	Functional requirement	Orthogonal
Data delivery	High	Yes	No
Propagation through interested peers	High	Yes	No
Churn-resilience	Low	No	Yes
Data-availability	Medium	No	Partially
Load-balancing	Medium	No	Partially
Scalability	Low	Yes	Yes
Efficient bandwidth utilisation	High	No	No
Low overhead	Medium	No	No

Table 5.1: A list of requirements

Chapter 6

Solution

In the previous chapter we have talked about the requirements that must be satisfied by a good solution for propagation protocol within DHTs. In this chapter we will describe our solution and how it meets the requirements described in Section 5.2. The solution must be able to propagate files of any size and take into account peers bandwidth capacity and utilise it efficiently. We will also present an overview of the components in our system and how they meet our functional and non-functional requirements. We will go through each non-functional requirement and discuss how components in our systems together meet them.

6.1 System components

There are many components in our systems. In the following sections we will describe the components. We will briefly describe a DHT, the overlay and consistency management protocols in DHTs.

6.1.1 DHT

Our solution for propagation in DHTs is not specific to any DHT implementation. Our aim is to provide a protocol that can be implemented on top of any DHT and efficiently utilise peer bandwidth to propagate large files quickly and balance the propagation load among peers. We assume a standard DHTs interface i.e. *put*, *get* and *remove*. Our focus is to improve the propagation protocol within a DHT without degrading the existing properties such as load-balancing, churn-resilience and availability etc.

6.1.2 Replica overlay

A standard DHT has no specific implementation for keeping track of replicas. As the number of stored files by every peer increases, a larger list of replicas has to be maintained. This causes more messages to be exchanged and increases the overhead. Keeping an overlay for each item is useful when propagating information in DHTs. For instance, a replica r,

must not be involved in propagation of data for file f, if r has no interest in f. This requirement can be met by creating a separate overlay for every file. However, this must not affect the scalability of the system, both in terms of peer in the system and number of overlays in the DHT.

It is no coincidence that we have talked about SpiderCast in Section 3.2 on page 12. Using SpiderCast we can create the required overlays and propagate files through these overlay to avoid involving uninterested peers. SpiderCast is especially beneficial for our purpose because it is fully distributed and resilient to failure. As a result it is scalable and load-balanced. These properties help us avoid degrading the churn-resilience and load-balancing of standard DHTs, if not improve them. We emphasise that our work uses an adaptation of the original SpiderCast described in Section 3.3 on page 14. We have included a brief description of the original SpiderCast in Section 3.2 on page 12.

SpiderCast has no central point of failure and is fully distributed. Using this overlay we can create and maintain a sub-overlay for each separate file. For instance, SpiderCast overlay in our implementations create sub-overlays for each file hosted by the peer. This overlay is used to propagate the files to partial replicas while allowing for partial replicas to exchange pieces without the *coordinator* intervening. This allows us to reduce the bandwidth required from the *coordinator* to propagate such files. Files can be propagated along these sub-overlays and exclude any peer not interested in the file.

Another beneficial aspect of SpiderCast is that it can create has greedy links by continuously connecting to peers with most shared interests and disconnecting from redundant links. This helps to maintain K-coverage while minimising the number of links that have to be kept up. By avoiding redundant links, we can avoid sending control messages and use peers' resources efficiently. This fits well with our approach to pessimistic bandwidth capacity. SpiderCast is also resilient to churn which makes it very suitable for our non-functional requirements as well as functional ones. We will describe some of the essential procedures of our adaptation of SpiderCast in Section 7.2.

6.1.3 Push-pull

We have briefly described some of the protocols used to manage consistency among replicas, their advantages and disadvantages and their ideal application for each protocol in Section 4.3 on page 20. We have implemented a solution that is very similar to the standard push-pull protocol described in section 7.3 on page 43. This protocol benefits from push and pull protocols while simultaneously avoiding their disadvantages. By using this protocol, we can exchange messages and notify peer about file creation, deletion or update events quickly. This enables replicas to react faster to an event, while enabling peers to pull the data when it is best suited for them. Another benefit is enabling peers to choose the source of data. As peers might have different bandwidth capacity, it might be best to request data from a source with superior capacity. We will describe a standard ver-

sion of push-pull protocol thoroughly before introducing our new protocol in Section 7.4 on page 45.

6.1.4 Our new protocol

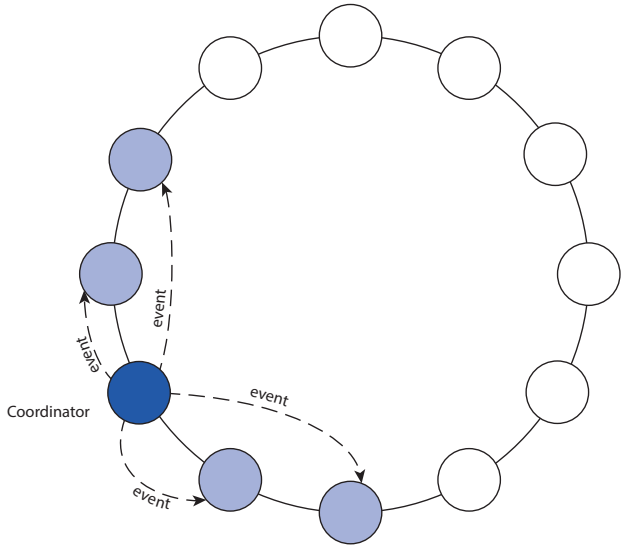

Figure 6.1: Step 1: The coordinator can propagate events using push messages.

The main motivation for our new protocol is to reduce the propagation load on the source as well as decrease the propagation delay. Our protocol can be used to propagate data quickly. Our approach is very simple. The basic concept of our protocol is to divide large data into pieces and propagate the pieces among interested peers. Instead of propagating entire files, partial-replicas can send and receive pieces of files. This allows partial-replicas to request pieces from each other during the propagation period. This results in a propagation technique similar to peer-to-peer manner, rather than a master-slave scheme.

In order to enable such propagation protocol, peers must notify each other of the pieces that are available locally due to reception of pieces. This is done by modifying the content of the push messages. Push messages are sent to interested peers and not any uninterested peers. Thanks to SpiderCast this can be done by simply using the SpiderCast overlays.

The second modification is related to how partial replicas pull data. Partial replicas that have received push messages from others, inspect them and check if there are any *interesting* pieces received by neighbours. We define interesting as pieces that have not to been received nor requested from others peers earlier. Partial replicas will then request piece(s) from the coordinator or any partial replica as illustrated in Figure 6.2. Although this can create a temporary hotspot for the coordinator, it will not last long. In this step, the probability of requesting from a master replicas is:

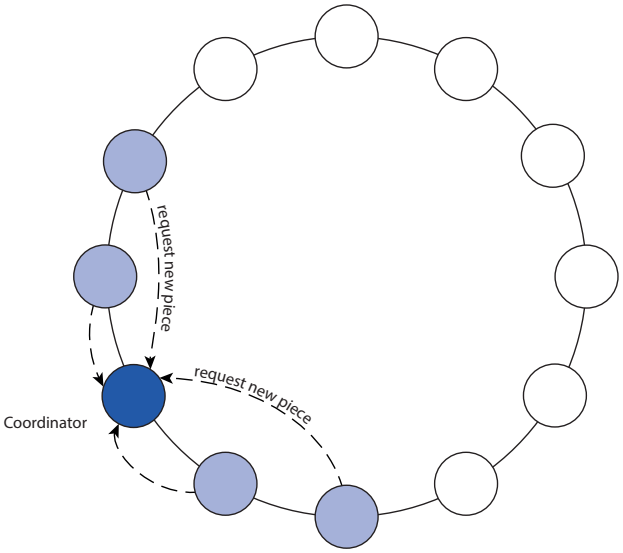

Figure 6.2: Step 2: Replicas request pieces of a file according to their policy.

The third step is started soon after the pieces requested from the *coordinator* replica arrive to the partial replicas. Partial replicas will obtain information about the available pieces after they exchange push messages. This push mechanism keeps replicas updated on the pieces received by other replicas. Requests can be sent to partial replicas as well as the *coordinator*.

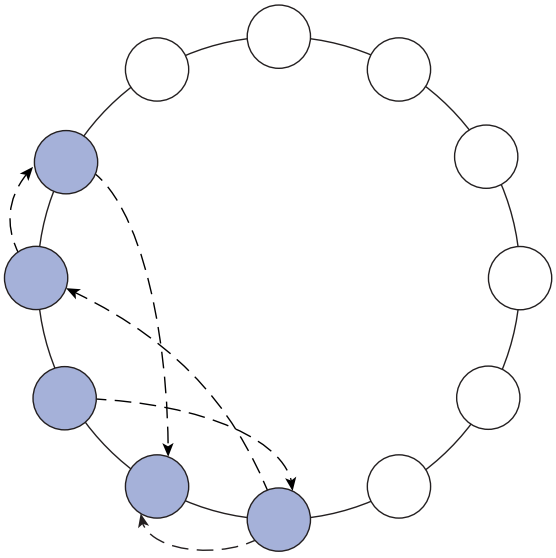

Figure 6.3: Step 3: replicas exchange piece among each other.

As more pieces become locally available in partial replicas, the possibility of a piece to be requested from the *coordinator* decreases. Currently in our implementation there is no bias for requesting from partial replicas.

However, in order to reduce the propagation load from the coordinator, it is beneficial to favour requesting from partial replicas.

The number of pieces requested can depend on peers' bandwidth. How these pieces are chosen is a criterion that has to be considered. It can be useful for replicas to receive the update pieces sequentially or randomly. The policy that selects pieces to be requested can be very application-specific. This policy may also depend on type of files stored, or the mechanism used to read that file. For instance, a DHT-client might want to receive pieces of a video file sequentially to watch the video while it is being downloaded. The same applies for audio files. However, other file types e.g. disk images or other large files, the order of which pieces are requested might not have any relevance. We will provide a pseudo-code for our protocol in Section 7.4. We will also evaluate this protocol and compare it with the standard push-pull based protocol as we believe they are the best baseline for testing. To avoid confusion in the evaluation process we will this protocol peer-to-peer propagation protocol (P2PPP).

6.2 Non-functional requirements

In this section we will go through the non-functional requirement that we have described earlier, and evaluate if our solution meets them.

6.2.1 Churn-resilience

Because DHTs such as Chord and Kademlia are already robust systems that handle churn well, we focus on avoiding the use of any algorithm that would degrade this robustness. As SpiderCast is fully distributed and robust, it fits very well in DHTs in terms of failure-tolerance. SpiderCast allows for configurable amount of fault tolerance as it allows connection to random peers to avoid network partitions.

Another design choice is anti-entropy protocol that was previously described in Section 4.3.3 on page 21. This protocol offers many desirable benefits. The push-pull based protocol benefits from lowering the overhead of the pull protocol while letting replicas request files when it is suitable for them, unlike the push protocol. Although this protocol introduces some overhead, it is insignificantly small. In addition, such protocols can be very useful in dynamic environments where churn is anticipated.

6.2.2 High data-availability

As we are focusing on offering high data-availability we are choosing *eventual consistency* [35]. This choice enables us to make data available at all times, even if users can see inconsistent data. Our aim is to increase availability by enabling DHT to replicate their data faster. If the DHT is mutable and updates are being propagated, then we wish to decrease the update propagation time.

Another aspect that contributes to data availability is the replication policy. By requiring a certain amount of available replicas, we can improve

data availability. The replication policy must also handle churn and maintain data availability by creating new replicas. This is also a standard component usually implemented and deployed in standard DHTs. We do not propose any modification of this component.

To summarise, high data-availability is achieved by using a standard replication policy. We assume that this policy handles churn and flash-crowds by creating more replicas as necessary. Although our protocol also benefits in DHTs with *strong consistency*, we have implemented a DHT with *eventual consistency* to increase availability.

6.2.3 Balancing the load of update propagation

Components that improve the load-balancing in the update propagation process are the SpiderCast overlay and our novel update propagation protocol which was previously introduced.

To avoid propagating files by involving all peers in the system, we create an overlay for each file. We use SpiderCast to solve this issue and distribute the propagation load among replicas. In other words, all replicas in the propagation process are interested in receiving the file themselves. Their interest in the update might be considered as an incentive for behaving properly in the propagation procedures, rather than maliciously. Our work also improves the balancing of propagation load in DHTs. The load on the *coordinator* replica is reduced by enabling partial replicas to connect and request pieces from each other.

To summarise, load-balancing is done by creating suitable overlays and enabling partial replicas to connect to each other. In addition, the replication policy is used to balance the request load on replicas and maintain the required quality of service.

6.2.4 Scalability with the number of files

Because we are distributing the workload in every component, our solution is very scalable. The underlying DHT implementation allows us to store many <*key*,*value*>-pairs while storing them in a load-balanced manner among peers. As the number of files grow, so does the number of overlays that must be created. We are assured by the fully distributed nature of SpiderCast that our solution will scale well. The scalability of SpiderCast has been evaluated thoroughly in the original paper[6].

Another component is our propagation protocol. As we reduce the propagation load on the coordinator, we expect our solution to scale better. We will compare our work with a standard push-pull based protocol, where entire files are pulled from the coordinator.

6.2.5 Efficient bandwidth utilisation

By utilising the underlying push-pull protocol, we can enable peers to request pieces with their bandwidth in mind. For instance, a peer with low-bandwidth might want to request one piece, while peers with higher

bandwidth capacity can request multiple pieces at once. In addition, each piece has a constant size e.g. 16 kB. Using these criteria, peers can calculate the number of pieces to request that suits their capacity and utilise their bandwidth efficiently. Utilising peers' bandwidth also enables our solution to reduce the necessary time needed to propagate files.

6.2.6 Low overhead of control messages

Peer bandwidth has to be used effectively as it can be sparse. As such we must choose carefully algorithms and protocols we use. Using SpiderCast we can connect to replicas with common interests together while keeping the number of active links relatively low. Reducing the number of links that have to be maintained, by itself eliminates unnecessary control messages used by e.g. TCP/IP protocols and improve the efficient utilisation peer bandwidth. The result is a reduction in overhead.

In addition, we are using push-pull based protocols to propagate files. This protocol introduces the least amount of overhead while at the same time enabling peers to request files pieces according to their bandwidth capacity. This flexibility also extends to file prioritisation we described earlier. For completeness sake, some alternative protocols for propagation were described in Section 4.3 on page 20.

Another method for reducing the number of control messages is by our choice of consistency type. We decided to use *eventual consistency* as it focuses on availability of data. This enables us to avoid the control messages used by consensus [23], locking or mutual exclusion algorithms. All of these algorithms can be used to offer *strong consistency*.

To summarise, our approach is to keep overhead low by using SpiderCast, which can select neighbours carefully and conservatively. Using *eventual consistency* as our consistency type also enables us to avoid some overhead control-messages. Push-pull protocols are used to limit the overhead and offer flexibility we require.

6.3 Summary

In this chapter we have described some of the components we assume are available in the DHT. This assumption is simply met as the components standard and may exist in some of the current implementations. Our functional requirements were discussed in Section 6.1.2 and with our new protocol. Finally, we have discussed how our non-functional requirements are met with our proposed solution.

Table 6.1 shows how our proposed components effect the requirements we defined earlier in Section 5.2.3. As our solution assumes that bitmaps are included in the push messages, the size of messages will increase. Although the overhead is very small, it will be one of our evaluate metrics.

Requirements	Improved / Weakened / No change
Data delivery	No change, all peers receive the files
Propagation through interested peers	Improved using SpiderCast
Churn-resilience	No change.
Data-availability	Improve by the reducing propagation time
Load-balancing	Improved by reducing the propagation load
Scalability	No change.
Efficient bandwidth utilisation	Improved. Slow-sender and slow-receivers can not delay propagation
Low overhead	Weakened by increasing the size of push messages

Table 6.1: A list of how our requirements were met using the solutions mentioned.

Chapter 7

Protocols for solution

7.1 Introduction

In this Chapter we will talk about some of the SpiderCast protocols as well as the standard push-pull protocol. In addition we will provide a pseudo-code for our protocol. Components such as replication policy, failure detection and failure recovery are not discussed in this chapter. However, we assume that such standard components are implemented. We emphasise that the SpiderCast described is our adaptation of it and not the original SpiderCast.

7.2 SpiderCast overlay

We have previously given a brief description of the SpiderCast overlay in Section 3.2 on page 12. In this section we will go through three essential aspects of our adaptation of SpiderCast in detail; connection, disconnection and neighbour maintenance procedures. We emphasise that the description that follows is an adaptation of SpiderCast implemented in [41] which again is an adaptation of the original algorithms which are described in [6].

7.2.1 Maintenance procedure

In the original SpiderCast, each node maintains a list of neighbours and items available locally, hereafter called hosted files. For each hosted file, a list of replicas is maintained using the SpiderCast protocol. Figure 7.1 illustrates both data structures.

Peers join the system with an empty list of neighbours. Neighbours are added using push messages and SpiderCast procedures. SpiderCast will add new neighbours until every file has reached K-coverage. K-coverage is defined as having K neighbours that are interested in a file. This means that for every hosted file there are at least K links to neighbours with the same file hosted. After all hosted files are K-covered, a procedure for removing redundant links is executed. Redundant links are defined as links which will not disrupt the K-coverage if removed.

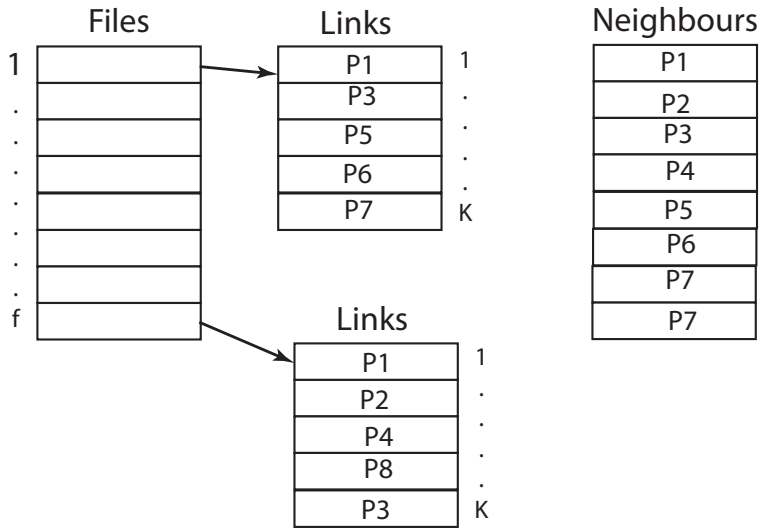

Figure 7.1: Some of the data structures in our system.

When a neighbour is added, their item lists are exchanged. For instance, when node n adds node p to its neighbour list, it exchanges item lists with node p. These lists are updated as new files are introduced to the DHT. Each node periodically executes three possible procedures: connection, disconnection, or drastic disconnection.

7.2.2 Connect procedure

This procedure is executed when a peer tries to add new neighbours and provide cover for all files hosted locally. This connectivity can be used for information propagation and consistency management among replicas. A node executing this procedure will select U uncover files and request a reference to replicas for these files. Each replica-reference is assigned a score. The score is calculated according to the number of links needed to provide K-coverage for file I. After scores are settled, the algorithm randomly selects candidates where the probability for a peer being selected is proportional to it's assigned score. This selection method favours peers with many common uncovered items, and disfavours peers with few common items.

For each chosen candidate, according to the random distribution mentioned above, a node requests a connection to be established. The candidate can accept or reject the connection request. The request can be rejected by the receiver. The rejection of the connection may be dependent on properties such as node degree, bandwidth, resource usage, etc.

7.2.3 Disconnect procedure

This procedure is executed when a node has achieved K-coverage for all its files using the connection procedure. The purpose if this procedure is to

remove redundant links without losing the *K*-coverage property. Similar to the connection procedure, each redundant link is assigned a redundancy score. This score is calculated according to the number of files it has in common. The redundant peers with the most files in common has zero probability of being selected. As the number of files in common decrease, so increases the chance of being selected for disconnection. Similarly a *disconnection*-request is sent. The request can be rejected by the receiver if disconnection leads to uncovered files on receiver's side.

7.2.4 Drastic disconnect

This procedure is executed when the number of neighbours exceed the maximum number for node degree. As we mentioned earlier, node degree is a term used to indicate the number of neighbours. A node executing this procedure will "repeatedly and unilaterally remove neighbours with fewest shared items until it reaches an acceptable number of neighbours."[41]. In other words the number of links are reduced to slightly lower than the maximum number of neighbours. Peers will then remove the files that are not *K*-covered. This is necessary to avoid exceeding the maximum number of neighbours as we attempt to provide *K*-coverage for every file. Similar to the previous procedures we assign each neighbour a score based on the number of shared files and the number of files that would be left uncovered in case this neighbour was removed.

7.2.5 SpiderCast Summary

We have showed how we are using SpiderCast to connect to peers with shared items. This is done by creating and maintaining a sub-overlay for each item. Since SpiderCast is fully distributed, it offers a load balanced and scalable solution for our purpose. We emphasise that our SpiderCast protocol is very similar to the one described in [41] with minor differences. Since we are using *pieces* instead of full files, we had to re-define what coverage really is. In our SpiderCast, coverage is defined as a peers hosting the same file both fully and partially. However, partial replicas must have at least one piece that is not locally available nor requested from others. In other words, we connect to peers that have partial and full copy of a file. However, peers with partial copies are not interesting to connect to if all the pieces hosted by that peer is already available locally. For instance, peer p might not find peer q interesting if all pieces q has are already locally available at peer p. In this instance, peer p might not connect to peer q, or may request to disconnect as the connection is marked as redundant.

7.3 Anti-entropy protocol

In this section we will briefly introduce a standard push-pull update propagation protocol and discuss the flaws in such a technique and how these flaws become significant in our solution.

In our system, anti-entropy protocol is used to notify other peers about events e.g. file creation, removal or update. By sending these notifications we can reduce the reaction time so that systems may react faster to changes. Also, by allowing peers to pull any piece of any file we provide flexibility for which piece to pull and whom to pull from. This way of pulling is flexible and allows developers to create their application-specific policies.

7.3.1 Standard push procedure

Anti-entropy protocols use push messages to notify about an event. The content of these messages is very application-specific. In our implementation these messages are used to notify a neighbour about the current state of hosted files. This is done by using a log. Logs are data structures used to indicate the history of files. When it is time for a new push period these logs are pushed to neighbours. To optimise this solution, each neighbour receives a different set of logs. This set is calculated based on the shared items. For instance, a neighbour n never receives a log about a file f, if n has no interest in f. Using these logs we can notify other replicas, while keeping communication overhead low simultaneously. Usually log entries might have fields such as: *item*, *event time* and possibly a *version*-field. In Section 7.4.3 on page 49 we will see how these fields are used and we will discuss what is missing and the fields that have to be added.

7.3.2 Standard pull procedure

As peers receive logs they perform a reconciliation process which is how replicas know what needs to be pulled and which peers to pull from. After this process is executed, the peers will pull the entire file content. There are many scenarios where this solution is not suitable. As peers request content from the *coordinator* of propagation and it is possible that the *coordinator* has a low-bandwidth connection, the propagation will be delayed. This delay is multiplied in cases where the number of interested peers is relatively large. As we saw in Section 2.1.1, it is possible for a peer to be *coordinator* for multiple files. This can also multiply the delay and cause a huge load imbalance in the system.

Another possibility is that slow-receiver peers that can cause a lot of delay. These peers can delay the propagation time if the coordinator propagates files in a sequential order. Another factor that needs to be considered is churn among *coordinators*. As the propagation time increases, so does the probability of a *coordinator* leaving the system during the propagation. Factors such as dynamicity of network, users' interests, file sizes and peers bandwidth have to be taken into account in order for a push-pull protocol to perform well within a DHT.

7.4 Our Approach

Earlier, we introduced a standard master-slave propagation scheme where the coordinator has to propagate the file content to every peer individually, using a push-pull protocol. We have showed that this scheme has some significant flaws in some likely scenarios in peer-to-peer systems e.g. low-bandwidth connections. In this section we will give a detailed description and pseudo code for our novel protocol for propagating data among replicas. Our focus is to improve (i) propagation latency, (ii) reduce the bandwidth sender has to offer when propagating in DHTs and (iii) utilise peers' bandwidth better by removing the bottleneck created by slow-senders. This latter is done by creating transfer slots. Transfer slots are can be thought of as a fraction of neighbour's bandwidth that is used to pull data. Using these slots we can allocate a fraction of neighbour's bandwidth to slow-sender peers while utilising the other transfer slots to request from other peers. Another advantage of using transfer slots is to enable application-specific policies that decide which files and pieces should be requested first. In other words, each transfer slots can have a specific policies that prioritise certain files or pieces. Lastly, using our protocol peers with low-bandwidth connections have the ability to postpone requesting a piece in order to participate in other parts of the DHT e.g. routing and consensus protocols etc.

In short, while master-slave propagation scheme disseminates the data by sending the data to each peer individually, we allow for *partial* replicas to exchange pieces among each other and reduce the propagation time and reduce the resources required from the *coordinator*.

7.4.1 Our approach: data structures

Each peer has multiple neighbours and files. Files hosted by a peer have variables similar to the ones described in Table 7.1. Furthermore, each file has a list of peers that are hosting the same file. We store a reference to these replicas in a data structure called *links*. The term *K*-coverage is translated as *K*-links for each file. In the following sections we will briefly introduce all the data-structures needed to understand the protocol we propose.

Hosted files

Each peer has a set of files it hosts. Files can be of arbitrary size, type and represent any kind of data. Peers must not only host the content, but also create and maintain some meta-data about these files. Table 7.1 shows some of the variables in the file-structure.

Links	Array of *links*. Each link is a neighbour hosting the same file.
Current map	Array of bits indicating the received pieces.
Requested map	Array of bits indicating which pieces are requested, but not yet received.
Modified bit	Boolean that is set if any changes have occurred since the last *push* iteration.

Table 7.1: Some variables of file data-structure.

Hosted file's links

As mentioned earlier we have a list of links for each file. Each link is a reference to a peer interested in the same file. Peers can either be a full replica, i.e. have a complete copy locally, or a partial replicas, i.e. only pieces of that file. For instance, a peer p that hosts file f, will keep a list l of other peers that also host the same file. The list may be partial and not include every peer hosting the same file. For each peer in that list, a set of information is stored locally. Table 7.2 shows some of the variables for each entry in list l.

Peer id	An identifier unique to each peer.
Map	An array that represents which pieces are available locally to that neighbour.

Table 7.2: Variables used to represent replicas in our system.

Event logs

As push messages arrive, they are appended to an internal log. These logs should be ordered by the happened-before relationship. There are three possible events that can be included log entries; add, remove and update. For instance, when peer p sends a log entry of type file addition, it means p is now a (possibly partial) replica for that file.

The set of logs propagated is dependent on the items that neighbour hosts as well as the logs we have sent earlier. This way neighbours never receive duplicate log entries or entries that were previously received. To conserve the communication costs, each log is sent to neighbours that share the same item. For instance, a log entry concerning item i will be sent only to neighbours that also host item i and not any other. Table 7.3 shows some variables needed to construct a log entry.

Peers exchange these logs in order to know about each other and use that information to determine which peers might be candidates for disconnection. The neighbours maintenance procedures were previously described in Section 7.2.

Peers being notified of events such as item additions, update their logs and create necessary data-structures according to that event. During the

Log type	Indicates if the entry describes an addition, removal or an update event.
Item id	The item involved in the event.
Map	An integer or an array representing which pieces are received.

Table 7.3: Some variables needed to construct a log entry

design of our protocol we have decided that receiving a new piece of a file should not be reflected as an entry in the event log. This helps us avoid having a log-entry of each received piece, which results in savings in term of resources such as storage and bandwidth. We consider this saving to be significant, when peers have low-bandwidth. In our protocol, as new pieces of a file is received, we set the *modified bit* introduced in Table 7.1. This bit indicates that new pieces have been received since the last push iteration. At the start of each push period, this bit is read and (if necessary) a singular log entry that represents the current state of an item is created. That entry is then appended to the local peer event log.

Neighbour

Each peer has to keep some meta-data about its neighbour. There are many variables included in the neighbour data-structure, but we will only describe the variables necessary to understand the protocol. As shown in Table 7.4, each peer keeps track of which logs have been received by each neighbour.

Another important variable for each neighbour is transfer slots. They help divide the available bandwidth capacity. These slots enable peers to request different pieces from different files simultaneously. This enables peers to prioritise different files and different pieces. It also allows for full use of bandwidth. Currently, the number of transfer slots is a configurable parameter. Lastly, we have enabled slots to request pieces according to a defined policy. Possible implementation are random, sequential, rarest-piece and most-frequent-piece policies.

Log index	An integer indicating the latest log received by that neighbour.
Push pending	An integer indicating how many log entries need to be sent on the next push period.
Slots	An array of transfer slots.

Table 7.4: Some variables kept on each neighbour

7.4.2 Our approach: system parameters

In this section we will describe some of the parameters we wanted to keep configurable. Some of them were later turned into a design decision based

on some preliminary testing.

Reactive pull

This parameter was first used to indicate if we should execute pull procedures immediately after the reception of logs during a push period or wait for the next pull iteration to start. If set, this parameter allows for immediate request of pieces from neighbours. That is, assuming the neighbours have some interesting pieces. Interesting pieces are defined as pieces not yet received and that are not yet requested from neighbours. The file pieces are requested only if there is an empty transfer slot. As result of some preliminary testing, we decided to always enable this parameter and change it to a design decision for our protocol.

There are two other types of reactive pulling. First, when a piece arrives through a transfer slot, we can either request another piece immediately or keep the transfer slots idle until the next pull iteration. The parameter is a boolean that, if set enables reactive pulling on successful reception of a piece.

Another similar parameter we decided to change to design decision is **reactive pull on slot timeout**. This parameter is similar. If enabled, a request is sent immediately after a request timed out. If disabled, we wait for the next pull iteration to occupy the transfer slot.

Number of transfer slots

This parameter is a simple integer that defines the number of transfer slots a peer has. In our current implementation all peers have the same number of slots. The number of slots for peers in the system is configurable and will be evaluated in our evaluation.

Node timeout

This parameter represents the amount of time we must wait before sending another request to an overloaded node. A peer that can not respond to a request because of its bandwidth or computational limitations will send an *overloaded* packet. When such a message is received we do not request from that peer for a configurable amount of time.

Slot policies

As previously mentioned, peers have a specific policy that may prioritise different files and different pieces. This policy can be divided into two different policies. A policy choosing which file to prioritise and another policy that chooses which pieces to prioritise. The file policy can be based on popularity of the file. The piece policy can be dependent of file type. For instance, videos are better requested sequentially, as they are mostly watched chronologically. However, both of these policies are very application specific.

7.4.3 Our approach: the protocol

In the next two sections we will describe our novel approach of push-pull protocol. We will start with the push procedures.

Push

In this section we assume that node n has started a push iteration and send its logs to peer p. Push procedures are called periodically. The next function initiates the push iteration. Before starting to push, log entries are appended to reflect the current state of peer n. Logs are then appended locally and exchanged with neighbours. The next function goes through the local log entries and only sends the logs tailored to neighbours interests. As the log filtering procedure is uninteresting, it is not included in the pseudo code.

```
1: procedure N.PUSH ITERATION
2:     prepare logs()
3:     for all n ∈ neighbours do
4:         if ( n.push pending ≠ 0) then
5:             send (<PUSH NOTIFY>, p)
6:         end if
7:     end for
8: end procedure
```

The next function is called when peer p receives a push message from peer n. The function checks if peer p is still a valid neighbour and processes the logs included in the push messages accordingly.

```
1: upon P.RECEIVING <PUSH NOTIFY>(neighbour n , Log[ ] Logs )
2:     confirm neighbour (n)
3:     for all entry ∈ Logs do
4:         process log(entry)
5:     end for
6:     send (<PUSH ACK>, n)
7: end upon
```

It is possible that a transmission error occurs during the transmission of push messages. This is why the *push pending* variable is not changed unless the reception of such messages is confirmed. To avoid inconsistencies and to ensure that neighbour p has received all logs sent to peer p a confirmation message is required. This is the main reason for this design.

```
1: upon N.RECEIVING <PUSH ACK>(neighbour p)
2:     confirm neighbour(p)
3:     decrease push pending(p)
4: end upon
```

Pull

In this section we will describe the pull procedures. The main focus of this protocol is to occupy all available slots i.e utilise all slots. Transfer slots send pull request according to a policy. The next function is executed both periodically and reactively, As described previously. The next function loops through all the hosted files and checks if any file is incomplete. We define an incomplete file as a file where all the pieces are not yet received. For each incomplete file, we loop through all neighbours hosting the same file. If an idle transfer slot is found, we use that slot to request a piece from that neighbour. In our pseudo code, peer n is initiating the pull procedures in order to request a piece from peer p.

```
1:  procedure N.START PULL ITERATION
2:      for all file ∈ n.hosted files do
3:          if (file is incomplete) then
4:              for all link ∈ file.links do
5:                  slot ⇐ find free slot (link.peer id)
6:                  piece ⇐ global piece policy (slot)
7:                  if ( slot is idle) then
8:                      send (<PULL REQUEST>, file, piece, slot)
9:                  end if
10:             end for
11:         end if
12:     end for
13: end procedure
```

To summarise what the next function does, we can take a look at possible events in this situation. If peer p has the requested piece, then a response OK message is sent along with the actual content. It is also possible for errors to occur. A possible, but not likely event, is that peer p has removed the file we requested. This can happen as a result of replication policy re-placing or removing replicas. In cases where peer p does not have the requested file, an error message is sent to peer n instead. Another possibility is that peer p does not have enough resources to serve peer n. As peer p receives a pull request from peer n the next procedures is called.

```
1:  upon P.RECEIVING<PULL REQUEST>(neighbour n, Request req)
2:      file ⇐ get hosted file (req.file id)
3:      if (file is available) then
4:          send (<DONT HAVE THAT FILE>, neighbour)
5:      end if
6:      piece ⇐ map.get (file.current map, req.piece)
7:      if ( available ) then
8:          send (<PULL RESPONSE>, piece, neighbour)
9:      end if
10: end upon
```

The results of the next function depends entirely on the type of message sent by peer p. We will describe the next function in terms of how we handle different message types. We divide the message types into the following scenarios:

***Response OK*:** This is our favourite case, where everything went as expected. The content of the requested piece is attached to this message. After the usual integrity and security checks, we check if the received piece is still *useful* before storing it locally. We say a piece is *useful* if it is still interesting to us. In some cases the replication policy can determine to remove a replica. In such cases the received piece is no longer useful to us. The *current map* and requested map for that item is updated to reflect the recently received response.

Finally we check if the received piece was the last piece. If so, we update the logs by appending an entry to the event log. That entry will then be pushed to interested neighbours on the next push iteration.

***DONT HAVE THAT FILE*:** This error message indicates that the destination does not host that item locally available any more, possibly because of an item removal. When receiving such a message we update our data structures. After the reception of this message the overlay for that file might change. Because our SpiderCast overlay choose the peers with most shared files to connect to, a reduction in shared files might invoke a disconnection procedure. SpiderCast neighbour maintenance procedures were earlier described in Section 7.2 on page 41.

DONT HAVE THAT PIECE Although this error message is very unlikely, we have created an error message to represent this error. As peer p pushes its map to peer n indicating which pieces are locally available to them, it should be impossible for peer n to request a piece that is not in map of peer p. But this is not a perfect world.

***CAN NOT ANSWER*:** The reception of this message indicates that we requested a piece from a peer that is overwhelmed with requests or limited in its' resources. After reception of this message, we keep the transfer slot idle for a certain amount of time. The idle period is a configurable parameter previously described in Section 7.4.2 as **Node timeout**. This case was omitted from our pseudo-code for simplicity sake.

To handle the different scenarios mentioned above, different operations have to be executed. However there are some operations that have to be executed *regardless* of these cases mentioned above. Operations such as (i) free (slot) and (ii) unset (*file*.requested map, *req*.piece) have to be executed regardless of message types.

In the next function peer n has to react to the type of message sent by peer p. The following is a pseudo code of such procedure.

```
1: upon N.RECEIVING <PULL RESPONSE>(neighbour p, response
   resp)
2:     piece ⇐ resp.piece
3:     resp type ⇐ resp.type
4:     version ⇐ resp.version
5:     file ⇐ get hosted file(response.file id)
6:     slot ⇐ resp.slot
7:     if (resp type = CAN NOT ANSWER) then
8:         reserve slot (slot, node timeout)
9:         unset (file.requested map , piece )
10:        return
11:    else if (resp type = DONT HAVE THAT FILE) then
12:        remove link (file, neighbour)
13:        free slot (slot)
14:    else if  (resp type = RESPONSE OK ) then
15:        set (file.current map, piece)
16:        if (file.current map is full ) then
17:            file.modified bit ⇐ false
18:        else
19:            file.modified bit ⇐ true
20:        end if
21:    end if
22:    unset (file.requested map , piece)
23:    free slot (slot)
24:    if (reactive pull) then
25:        file id , piece ⇐ slot policy(slot)
26:        file ⇐ get hosted file (file id)
27:        send (<PULL REQUEST>, file, piece, slot)
28:    end if
29: end upon
```

It is possible that the pull requests times out. Possible causes are that the request was not successfully sent or possibly that the response did not arrive successfully. However, another possibility is that peer p has left the system due to churn or other causes. In the next procedure peer n's request has timed out.

```
1: upon N.TIMEOUT <PULL REQUEST>(neighbour p, request req)
2:     slot ⇐ req.slot
3:     file ⇐ get hosted file (req.file id)
4:     unset (file.requested map, req.piece)
5:     free slot (slot)
6:     if (reactive pull) then
7:         file id, piece ⇐ global slot policy (slot)
8:         file ⇐ get hosted file (file id)
9:         send (<PUSH REQUEST>, file, piece, slot)
10:    end if
11: end upon
```

7.4.4 BitTorrent

The concept of dividing files into pieces is not novel, and has been used previously in BitTorrent for sharing large files. However the similarities end there. While BitTorrent cares about notions of fairness, tit-for-tat and auctions, we do not. We only consider the performance and throughput of the protocol and do not implement any fairness or tit-for-tat notions from BitTorrent. Our aim is to replicate large file in DHTs. Our focus is three criteria. (i) To quickly propagate large files in DHTs. (ii) Use peers bandwidth efficiently while taking into account slow sender and slow receivers. (iii) reducing the propagation load from the sender.

7.5 Summary

In this chapter we have discussed several protocols in our DHT. As DHT-routing is a known protocol, we did not describe it in this chapter. But, we did discuss The most important SpiderCast procedures. We also briefly described the standard push-pull protocol which will be used as a baseline in our evaluation. Finally, we have provided a generic and simple pseudo code for our protocol. We emphasise that the same protocol can be used to propagate updates by adding some versioning logic to distinguish between different version of files etc.

Part III

Evaluation

Chapter 8

Simulator

Recently some peer-to-peer simulators have been launched such as PeerSim [30]. PeerSim started as a EU research projects BISON (Biology-inspired techniques for self-organization in dynamic networks) and DELIS (Dynamically evolving large-scale information systems). Although PeerSim has become a popular simulator for peer-to-peer networks, it is implemented in Java and requires significant amount of memory. This was one of the main reasons for creating a new simulator for collaborative Wikipedia hosting in [41]. We will make use of the same simulator used in [41] because of its efficient memory usage. As a result we are able to run experiments with larger workloads and longer periods. The simulator is event-based and deterministic. In this chapter we will describe the simulator we are using to simulate the DHT events we have discussed earlier. As there are many components to the simulator, we will briefly describe them individually in the next sections.

8.1 Simulator architecture

The simulator is programmed in C. It is also memory efficient, very configurable and can be compiled on 64 and 32-bit machines using a GNU C compiler. The simulator has a centralized implementation, meaning it has global knowledge of the system. This implementation can help in gaining great insight into the aspects such as number of messages between peers, bandwidth usage, etc.

8.1.1 Simulator time

The simulator has defined cycles. Each cycle represents a second in a real-world. The coarse definition of time in the simulator is very beneficial for simplicity, understanding the simulator and further development on the simulator. It also helps with simulating longer periods of time e.g. months in the real-world. The same coarse definition is also a drawback. As shown in Figure 8.1, for each peer in the system no more than one transmission takes place per second. If the transmission takes less than one second, then the rest of the cycle is not utilised to transmit the next message. Similarly,

if transmissions end at the start of the cycle, the rest of the cycle will not be utilised to transmit the next message. Instead, the next message will be transmitted at the next cycle.

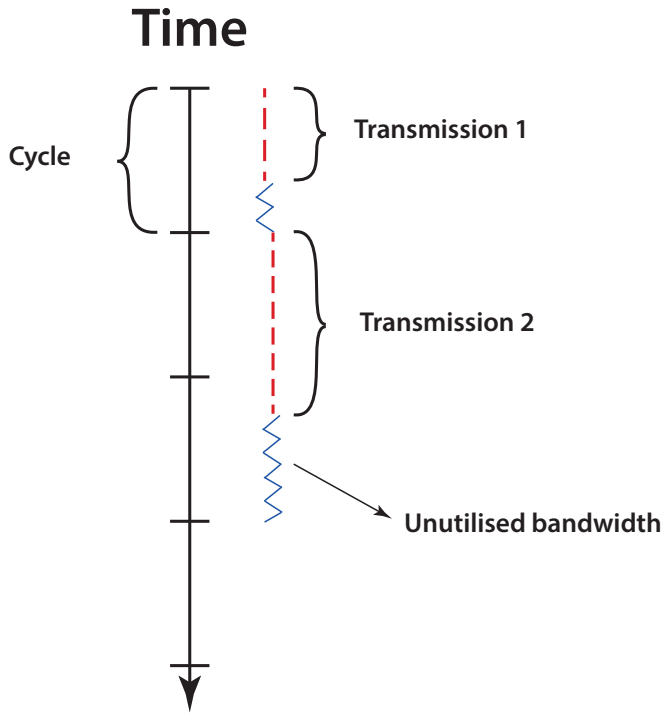

Figure 8.1: The effects of coarse definition of time in the simulator.

Although a small drawback this can easily be handled by defining the each piece according to the bandwidth capacity in order to utilise it effectively. We will attempt to avoid this issue by selecting piece sizes according to the bandwidth and vice versa. Although small messages still use an entire cycle, we believe the inaccuracy is relatively small and insignificant.

8.1.2 Limitations of simulator

The simulator does not simulate computations resources. We assume that the underlying DHT-implementation does not require any abnormal processing power. As this is true for Chord and Kademlia, two of the most known DHT-implementations, we believe this is a reasonable assumption. In addition, SpiderCast does not require any heavy algorithms. As shown in our pseudo-code, our protocol is not computationally heavy. Although, there are some possibilities for optimisation. In short, our simulator does not simulate the computation time of peers. This helps focus the monitoring procedures to more important aspects e.g. bandwidth. In addition, peers may have different processing resources. These resources may be utilised by other processes. We avoid such complexities by not monitoring the processing power and memory usage. Although the simulator does allow to simulating the storage space for peers in the

system, we will not make use of this aspect of the simulator as the assume that peers have enough storage space to store the files propagated. It is possible to implement a replication policy that places replicas according to their storage capacity. In short, peers processing power, memory usage and storage space are uninteresting aspects to us. As the simulator has global knowledge, it can monitor the connectivity of peers, files and accurately monitor the number and size of messages sent during the simulation period. For more technical details about the simulator see [41].

8.1.3 Input files

The most important types of input files for the simulator are:

Files in the system must be defined in a separate file before execution starts.

Peers are especially important as they are providing the service. The number of peers, their bandwidth are important aspect of our evaluation and important input for our simulator.

Events are also important as they define when an event such as updates or file creations occur.

Churn is also useful to see how resilient DHT and other components are to mass failure. The churn events in the file only represent event such as peers leaving and joining the system.

Parameters This is a large file that includes every parameter value needed for the simulator to function. As the simulator is very configurable, there are many parameters that are not interesting for our purpose. We will focus only on the parameters that are interesting.

There are other files that must be input for the simulator to work. However, we have focused on the files that are important for our thesis. Another very important aspect of the simulator is the parameters for the simulator. The simulator will result in different results as the parameters are changed. There are too many parameters to include here. However, we will provide a set of parameters for each components as we describe them in the next sections.

8.1.4 Output files

The simulator has a number of components that are not interesting to us and allows for monitoring and logging of these components. We have turned off logging of these components to avoid uninteresting results. However, the three most important output files than we measure are:

Delay: The simulator logs the time (cycle) in which the *coordinator* has received a full copy of the file. This is done for all files. Each peer that has successfully pulled a full copy of the file notifies the simulator. The time (cycle) where the last notification was received is logged by the simulator. As the simulation ends, this log is written to a file.

Bandwidth consumption: Although the simulator previously allowed for bandwidth monitoring, we had to create a monitor specifically for our purpose. This was done, because we wish to monitor the bandwidth consumed by push and pull procedures individually and for each item. The bandwidth monitor works as such. For each pull or push message, the header size of the message is ignored as it is small and considered insignificant. Instead, we focus on the payload sent. Each pull and push message sent is monitored and it's payload logged. This applies to all peers and items. At the end of the simulation these log are written out as files.

8.2 Simulator components

8.2.1 DHT

As our work is independent of any specific implementation of DHT, we only need to simulate a DHT-lookup procedure instead of actual DHT-routing functionality e.g. Chord. Chord routing technique was described in Section 2.1.1 on page 9. Since the simulator has global knowledge about the system, the DHT-lookup is implemented as a function that returns pointers to the peers that would have been the result of an actual DHT-lookup. The result of the DHT-lookup is a set of *peer id*s that are closest to a given key.The notion of closeness in our implementation of is done by consistent hashing, the same technique used in Chord any many others systems such as Amazon's Dynamo [11] and Facebook's Cassandra [22]. As different DHT-implementation may have different failure recovery techniques implemented, the simulator only considers the live nodes when responding to the DHT-lookup to avoid dependency on specific failure recovery techniques.

To simulate the bandwidth consumed by the DHT-lookup correctly, we have simulated a large message exchange between the DHT-lookup initiator and the destination. The size of the large message is the sum of all messages sent for each hop in a DHT-lookup. This helps make the simulator faster in execution time, which helps us simulate longer period of real-world events. The delay for looking up peers in the DHT is not simulated. However, this is not an issue since our baselines make use of the same DHT-lookup, thus the results will be valid when comparing the baselines. In addition, our protocol does not require any DHT-lookup to propagate files. This is done by the replication policy, which is used for both our protocol and the standard master-slave propagation protocol. During our sensitivity analysis, the parameters changed do not affect the number of DHT-lookups. However when more replicas are created, we have changed the number of peers returned, when a DHT-lookup is initiated. We will discuss this parameter more thoroughly in Section 9.5. In short, the DHT-lookup implemented does not affect the validity of our results and helps us improve the execution-time and simulate real-world events faster.

8.2.2 Replication policy

In order to enable replication and evaluate our thesis a replication policy is required. These policies are very application-specific, as it depends on the focus of system designers. The replication policy can be focused toward providing data-redundancy, data-availability, reduced response time, load-balancing etc. For instance, Cassandra [22] allows for replication according to different policies. As different focuses would require different implementations, we have not implemented such dynamic policy with these factors in mind. Instead of a dynamic replication policy, we have created a static replication policy that would simulate the worst case events. Using our static policy we can simulate the worst cases for a dynamic replication policy.

Our implemented policy does a DHT-lookup and chooses a random peer as the *coordinator* and pushes some meta-data to the other peers notifying them about the file-creation event. Each peer will then connect to the *coordinator*, as that is the only peer that has interesting pieces. The file will then be propagated according to our protocol. Our simulator logs the time when the coordinator received the file and when all peers have a full copy of the file. The mentioned policy simulates the cases where a configurable number of replicas must be created at once. For instance, as a new file is added into the DHT it must be replicated instantly. Another possible scenario could occur under churn. Under churn, the replication policy must create a number of replicas instantly to remain within the required data-redundancy or response time. The same actions are required as files such as videos become extremely popular. To reduce the load on replicas and improve response time, numerous replicas must be created as fast as possible. In all cases files need to be propagated in the DHT, which requires a quick, load-balanced propagation protocol. We emphasise that our work is completely orthogonal to replication policy and can be used with any policy.

8.2.3 SpiderCast

Our adaptation of SpiderCast was thoroughly described earlier in Section 3.3. We have implemented these changes in our simulator successfully. However, we will not evaluate this protocol individually since a thorough evaluation can be found in [6]. As SpiderCast is part of our system, its performance will be indirectly evaluated. Figure 8.2 represents three possible overlays created for each item. The Chord ring is meant solely for illustration as neither SpiderCast nor our work depends on a Chord-implementation.

8.2.4 Summary

In this chapter we have described the components required in our evaluation and how they are simulated. As replication policies are very application dependent, we have implemented a policy that simulates the

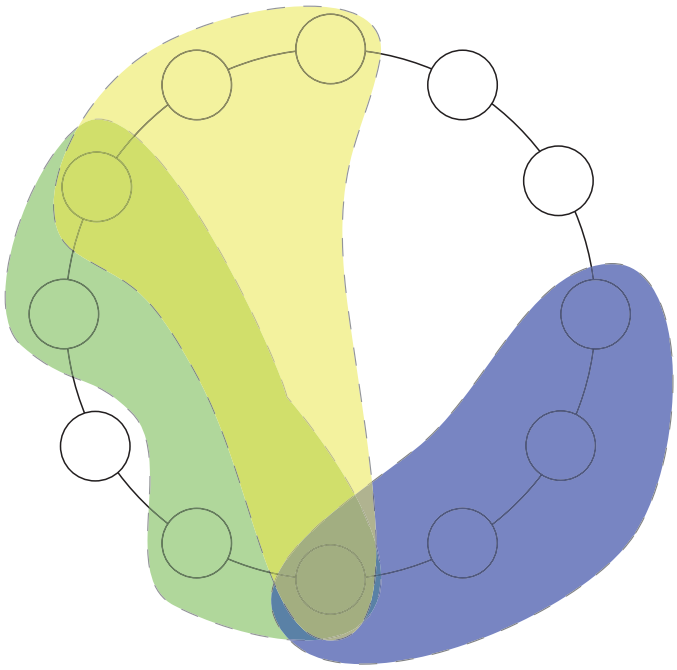

Figure 8.2: An overview of three possible overlays created by SpiderCast on top of Chord.

worst scenario for a dynamic replication policy. In this scenario numerous replicas have to be created instantly. This is done by propagating the files to peers within the DHT. Such scenarios can occur as files are inserted into the DHT, updated, possibly due to churn or flash-crowds. We emphasise that our work in completely orthogonal to components such as replication policy.

Figure 8.3 shows the some of the components in our simulator and how it could be possible to use our propagation protocol for different purposes. By adding some simple version-logic it is possible to use our protocol to propagates large updates in DHTs.

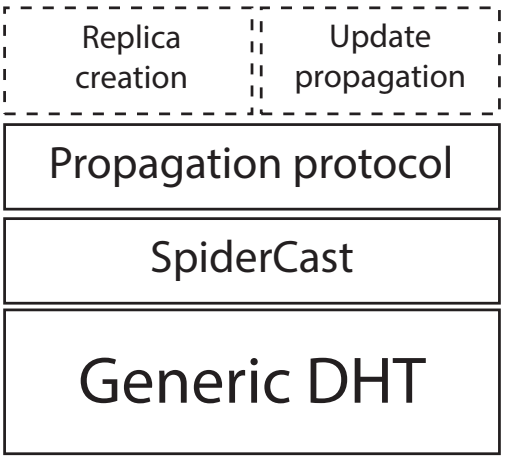

Figure 8.3: An overview of the simulator components.

Chapter 9

Setup

In this Chapter we will show the results of the experiments we have simulated and interpret the results. We will focus our evaluation on our protocol's performance and overhead and compare the results with a standard master-slave protocol. Our goal in the evaluation is to see how fast our protocol is able to propagate files within DHTs while at the same time how much overhead is introduced by our protocol. As there are many parameters in our simulator, we can not evaluate all combinations of the parameters. However, we will run a sensitivity analysis and see how our metrics are affected by changes to a single parameter, given that there are default values for our experiments.

The rest of the chapter is as follows. In Section 9.1 describe the scope of our evaluation. We will provide a detailed description of our evaluation metrics in Section 9.2 and use the dataset described in Section 9.3 in our evaluation. We will provide some examples for suitable baselines in Section 9.4. In section 9.5 we will discuss some of the parameters in our system and reason about their default values.

9.1 Scope of evaluation

As there are many possible aspects to monitor in our system, we have to limit we wish to evaluate and why that is an interesting aspect to evaluate. It would be interesting to see how efficient and quick SpiderCast can create an overlay. As there are different methods and many practical and synthetic traces for evaluating overlays we consider the replica overlays outside the scope. A thorough evaluation of the original SpiderCast can be found here [6]. An evaluation of SpiderCast in a practical peer-to-peer system can be found here [41]. However, by evaluating propagation time, we can indirectly see the effects of SpiderCast as a new replica overlay is created and used to propagate pieces among peers.

An aspect that is interesting to study is the load-balancing in context of propagation. Generally, load-balancing is an important metrics to study. However it can be defined and measured in different ways. Balancing the load of users' read-request between available resources is very important for making systems more stable and reliable. However, balancing

users' write-requests might require different actions than balancing read-requests.

Similarly, defining and measuring load-balancing in the context of propagation can be difficult. Since our protocol reduces the propagation load on the *coordinator*, it would be interesting to see how effect of our protocol is in terms of propagation-load balancing. However, measuring propagation load can be difficult and requires a lot of time and effort. Due to time constraints we will prioritise the more important aspects of the evaluation. In our evaluation we will only consider the metrics, propagation time and overhead introduced by our protocol.

Another (less) interesting technical aspect to study is the effects of message timeouts and frequency of messages might have on the system. As there are different messages in our systems, changing the waiting period of messages can effect many components and protocols in our simulation. Similarly, the frequency of messages can have an impact on the protocols. As such we will not run sensitivity analysis on such small changes in our system, and focus on the essential parts i.e our propagation protocol.

9.2 Evaluation metrics

We have chosen our metrics to evaluate the performance of our propagation protocol. There are two evaluation-metrics for evaluation that will be considered.

9.2.1 Propagation time

To propagate copies of a file, the *coordinator* has to propagate the file to other peers. The time it takes from the coordinator starts the propagation to the time all peers have a full copy of that file, is defined as the propagation time. We say that propagation stops when all peers have a full copy and they are all consistent. We will measure that time and see how it is influenced as we change some parameters. Figure 9.1 illustrates a time-line for a propagation process and how we have defined the end of propagation process. In short, the propagation does not stop until all transfers are completed.

Figure 9.1: The definition of propagation time in our system.

9.2.2 Overhead

An interesting metrics that we will evaluate is how much overhead our new protocol introduces. But first, we have to define what we consider as overhead.

As described earlier in Chapter 7, there are many protocols in our system that contribute in the overhead by sending control messages. For instance, ping-messages can be considered overhead even though they can contribute in making the system more stable and reliable by detecting some failures. DHT-messages and DHT-hops required to serve a DHT-lookup can be considered overhead. Control messages introduced by the replica overlay, SpiderCast in our case, can be thought of as overhead. Due to the number of protocols and multiple control messages for each of them we will not monitor all control messages being sent.

As our baseline is a standard master-slave protocol, the same protocols are used to provide the same functionality. Our protocol makes use of the same protocols in order to function. Table 9.1 shows a comparison of the protocols used in our evaluation. As our the same failure detection, DHT and overlay protocols, their control-messages and overhead is uninteresting. Moreover, our propagation protocol does not modify any of the protocols. In addition, our protocol does not require additional DHT-lookups compared to our baseline. This guarantees that excluding such protocol does not affect the validity of our overhead measurement.

Protocol	**Our work**	**Standard master-slave protocol**
Failure-detection overhead	No changes.	No changes.
DHT overhead	No changes.	No changes.
SpiderCast overhead	No changes.	No changes.
Push overhead	Push message with maps	Push messages without maps
Pull overhead	Pull-request for each piece	One pull-request.

Table 9.1: A comparison between our protocol and the standard master-slave protocol

Table 9.1 shows the differences that were introduced by our protocol. Obviously, in most cases, our protocol requires more pull-request compared to our baseline. However, due to the size of the header for pull-messages[1], we consider them negligible and not worth monitoring. Another change introduced by our protocol is the size of push-messages. Our protocol does not require any changes in the frequency of push messages, but only the size of such messages. In our protocol, the push messages sent have to include the necessary information to represent the current state of received pieces for a peer participating in the propagation. This information is included in push-message by using bitmaps[2]. As discussed earlier, the push-message headers are also of insignificant size, thus we will focus of the size of the bitmaps that have to be included in push-messages.

[1] Few Bytes

[2] An array of bits,were a '1' represents a true-value, and '0' represent a false-value.

To measure overhead we consider what our protocol has introduced, namely the bitmaps included in push-messages. In our implementation, we have put these maps as the payload of the push message. This way, we can measure overhead is by simply monitoring the bandwidth consumed by transmitting the payload of push-messages. This is done on a per file basis. By monitoring how much bandwidth push procedures consume for each file, we can accurately measure the necessary amount of overhead introduced by push procedures during the propagation of a file. We use the bandwidth consumed by the push protocol to study the overhead introduced by our protocol. Since the standard master-slave protocol does not include any maps in the push messages, there is no payload to monitor. Our measurement will accurately measure the relative overhead consumption compared to the standard master-slave protocol.

9.3 Dataset

Recently a collaborative Wikipedia hosting has been proposed in [41]. The architecture uses a DHT as a cache for popular Wikipedia files. This includes pages and multimedia content (mostly images). We have used the Wikipedia trace from [41]. The dataset was gathered by sampling the HTTP-requests to Wikipedia servers for a month. We believe that this dataset can simulate how well our protocol performs when the DHT has a mixture of small and large files. Using this dataset we can gain insight into thresholds where our protocol performs as well as the standard protocol.

The trace is divided into two files. The first includes information about file types, sizes, creation date etc. The second file has information about events that occurred on these files e.g. creation or update time. In order to accurately simulate the insertion of a relatively large file in a DHT, we have filtered out the Wikipedia-page entries as they are very small and produce a lot of uninteresting data due to their small sizes and their size distribution. We have also filtered out the images that were uploaded before our the sampling period started. The dataset left was the images that were uploaded to Wikipedia servers *during* the sampling period. We believe this is a dataset that can accurately simulate the real-world events occurred on Wikipedia server. We will use this dataset to simulate the events of images being inserted into a DHT. The event of an uploaded file to Wikipedia servers can be viewed as a file inserted into a DHT in case of peer-to-peer web hosting system. As files are inserted, they have to be replicated across the DHT according to our replication policy.

There are more than 8400 images in our dataset. Figure 9.2 is a histogram of the sizes in our dataset. As can be seen in the figure, the images are mostly between 10 KB and 100 KB. However, there are large images e.g. more than 18 MB and also small images e.g. 44 Bytes. The x-axis in Figure 9.2 is exponential to improve readability.

To provide more insight into the Wikipedia trace, we have included the frequency of events that occurred in the trace. Figure 9.3 shows a histogram of events occurred during each hour of sampling period. Spikes in the

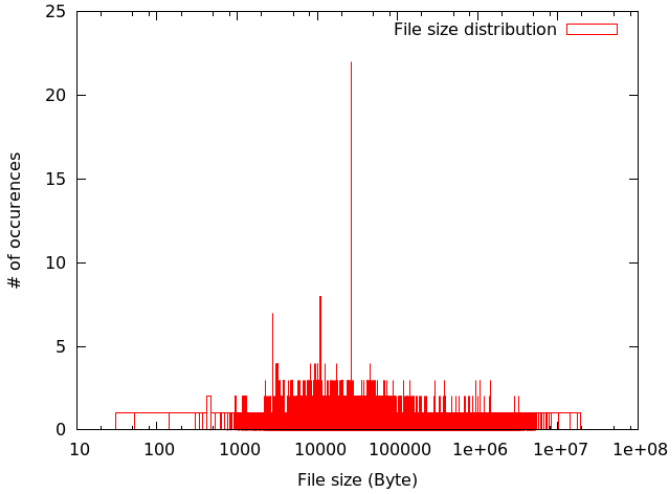

Figure 9.2: Distribution of files size in our dataset.

graph is caused by increased number of images uploaded, possibly due to Wikipedia-entry editing. Because simulating the entire trace is resource and time consuming, we have limited our simulations to simulate the first ten days of our trace. The figures provided only included the files and events that occurred during the first ten days of our trace. We believe that this does not affect the evaluation process negatively, as there were more than 8400 files uploaded that span from 44 Bytes to more than 18 MB.

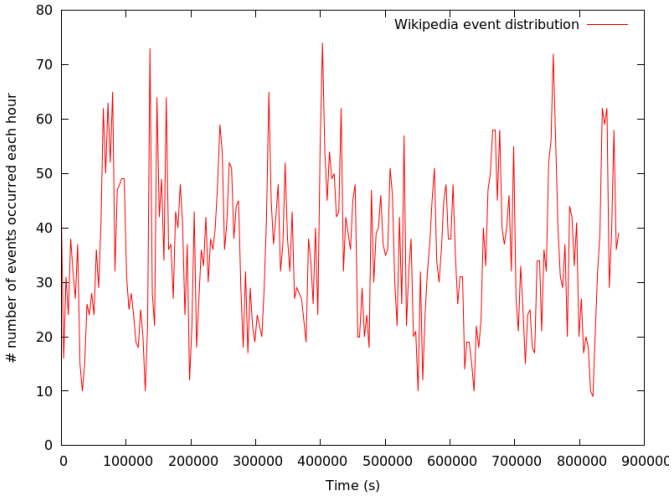

Figure 9.3: The distribution of events occurred in our dataset.

9.4 Baseline

To avoid confusion between the protocols we will refer to our work that were described in Section 7.4.3 to peer-to-peer propagation protocol (P2PPP). In this section we will compare P2PPP to two baselines.

9.4.1 Standard master-slave baseline

Our main baseline for comparison will be the standard master-slave protocol. This protocol was previously introduced in Section 4.1. This protocol can be used to synchronise among loosely synchronised replicas. We will call this protocol as the master-slave (MS) protocol to avoid confusion.

9.4.2 Lower-bound baseline

Another baseline that is interesting to study is the lowest possible propagation time. In this baseline peers use their entire bandwidth only to propagate files. Moreover, all peers will start utilising their entire bandwidth immediately after the *coordinator* has received the file[3]. This is an ideal scenario where the only bottleneck is the bandwidth capacity. This is practically unlikely, because peers must also participate in other parts of the DHT e.g. failure detection, overlay maintenance etc. However, we will use this baseline to show the true performance of collaborative propagation. The lower-bound propagation time is calculated based on peer's bandwidth. For instance, ideally a peer with 100 KB/s bandwidth capacity could receive a 100 KB file in a second. This baseline is not completely accurate as peers participate in other parts of the system. In addition, the coarse definition of time in the simulator, could increase the propagation time as shown in Figure 8.1 on page 58.

9.5 Parameters and default values

As there are too many parameters in the system, we will not evaluate every combination of parameters since it will take a lot of time and result in a lot of uninteresting information. To avoid this issue, we will evaluate some parameters individually and analyse their influence over the evaluation metrics. Table 9.2 illustrates the parameters that are most essential to our system. Some of the parameters were previously described in Section 7.4.2 on page 47.

Peers are less likely to contribute huge amounts of bandwidth if they have no incentive to do so. However, it is possible to incentivise peers to join our systems if the amount of bandwidth they have to contribute is low. As such we assume that peers only contribute a small portion of their bandwidth capacity. We will assume that peers contribute approximately 30 KB/s to our system. Previously, values such as 200 kbps (25 KB/s) were used in [41] for evaluation purposes. Due to technological advances and increases bandwidth capacity we believe 30 KB/s is fit for a default value. Moreover, during our evaluation we will run simulations where peers have contributed 15, 30, 50 and 75 KB/s bandwidth to our system. These values are used for both in-bandwidth and out-bandwidth. In other words, in all our experiments the fraction of $\frac{bw_{in}}{bw_{out}}$ is set to 1. We believe these values

[3]Which implies infinite *coordinator* bandwidth.

System parameters	Default values
Number of peers	10000 peers
Bandwidth homogeneity among peers	Yes
Reactive pull on successful piece reception	Yes
Reactive pull on failed piece reception	Yes
Reactive pull after push iteration	Yes
Node timeout	50 Second
Replication parameters	
Number of replicas to create	7 replicas
Number of peers returned in DHT-lookup	10 peers
Churn Parameters	
Mean offline time	None
Maximum peer availability	None
Availability skewness	None
Peer parameters	
Number of transfer slots for peers	2 slots
Peer bandwidth	30 KB/s
Size of pieces in the system	30 KB
Transfer slot request policies	Random policy
Frequency of pull procedures	100 Seconds
Frequency of push procedures	30 Seconds
SpiderCast parameters	
Neighbour selection method	Greedy
Desired neighbour for each file	3 neighbours
Frequency of maintenance procedure	3 Mins.
Maximum number of consecutive executions of the connection procedure	15 times

Table 9.2: Some important parameters for our evaluation and their default values.

are can provide deep insight into our protocol due to the wide range of values. As we are running thorough sensitivity analysis on the effects of bandwidth contribution is our system, we will not evaluate our system when bandwidth contributions are skewed. We will thus consider only homogeneous bandwidth contribution for all peers in the system. This can provide important insight into how our protocol performs when *both* senders and receivers have low-bandwidth. In our simulation there are 10000 peers in the system as that is a standard value used in [41].

We will use the dataset discussed in Section 9.3. We will furthermore enable reactive pull described in Section 7.4.2 and consider it a design decision because waiting for the next pull iteration to request a piece is simply not a good solution. Node timeout is set to 50 seconds. This parameter was previously described in Section 7.4.2. If a peer gets more requests than it could handle, it send a specific error-message to reflect that

state. Peers receiving such messages will have to wait for a configurable amount of time. We set this period to 50 seconds as we assume that it is enough time for "the storm to pass" and peers go back to their normal state. By running some initial tests, we have confirmed that this parameter does not effect our evaluation metrics and can be considered negligible. Moreover, analysing message timeouts are outside the scope of the evaluation.

As we are simulating the worst case scenarios we set the replication factor to 7. In other words, there are 7 peers in the DHT that will receive a full copy. This is done solely by the *coordinator*. As we are focusing on high data-availability and flash-crowd handling, we pessimistically imagine that a large image is uploaded due to editing of a popular Wikipedia entry. In such cases it is feasible to place as many as 7 replicas in the DHT, immediately. We will also run simulations where 5 copies of files are propagated. This value was previously used in the evaluation of [34]. Moreover, as many as 10 replicas could be created in context of social networks where user profiles have to be replicated in order to keep user's profile available even when the user goes offline. We speculate that these numbers are suitable for internet-scale peer-to-peer systems.

As the simulator is deterministic, a DHT-lookup will result in the same peers being returned. In order to gain a deeper understanding and avoid having a single experiment where a *lucky* coordinator has been chosen, we have decided to return 10 peers as our default size for the list of peers that are returned by the DHT-lookup. By randomly choosing a coordinator from this list, we can guarantee that our results do not reflect the *lucky* case where a good coordinator has been chosen to propagation the file. Moreover, we will run each simulation ten times with different random-generator seeds. This will make help in making the *lucky* choice of coordinator highly improbable. This is because it is highly improbable to randomly choose a lucky coordinator from a list to ten peers returned by the DHT-lookup in ten different executions, where ten different seeds are used to generate random values.

Although our reactive pull mechanism ensures effective utilisation of available bandwidth, the simulators coarse time definition limits our choice of pieces sizes. In order for us to simulate a correct bandwidth usage, we have to choose piece sizes according to the available bandwidth. For this reason we have set piece sizes to 30 KB. In addition, to show the parallelism of our protocol we have chosen to assign each neighbour 2 transfer slots as a starting point. These slots are used to request pieces from our neighbours. During our evaluation we will change the number of transfer slots for each neighbour and see the effects of these changes on our evaluation metrics. Another parameter is the policy by which peers request pieces or prioritise files. We have created a policy that identifies the interesting pieces and randomly picks an interesting piece to request. The only benefit of using such slots is to prioritise files, pieces or possibly neighbours. This can be very application-specific. As such we will not evaluate this parameter. A random slots policy is implemented for the purposes of our evaluation.

The frequency of pull and push procedures also has to be evaluated.

We will use the same frequency for pull messages as used in [41]. We will set the default frequency for push messages to 30 seconds. In other words, each 30 seconds the push-procedures are executed. Due to the scope of our evaluation frequency of messages are not evaluated. However, for fairness sake the same values are used in our standard master-slave baseline.

To keep overhead low and minimise the number of connection for each peers, we use the greedy method for neighbour selection in SpiderCast. In the original SpiderCast paper [6] it was noted that in most practical workloads connectivity could be achieved by having 3 greedily selected neighbours. As our purpose is also to achieve connectivity while minimising overhead, we will use the same values. We will also execute link maintenance procedures each 3 minutes and set the maximum consecutive execution of SpiderCast neighbour maintenance procedure to 15. These values were previously used in [41].

9.6 Churn experiments

As churn can be expected in peer-to-peer systems, we also evaluate our protocol under churn. We assume that peers will remember the data they had before leaving the system. Because there are different models for churn, we have defined several parameters to represent churn in our system. In all our churn experiments the default values in Table 9.2 were used.

Offline time represents the period for which peers go offline each time they leave the system. For instance, peer p goes offline for two hours each time it leaves the system. We will use mean offline time as one of our churn parameters. In addition we can have peers that are available for a percentage of the simulation duration. We call this metrics peer's availability. A peer with 50% availability will be online during half of the simulation period. To describe the churn in a more advanced way, we can enable peer's availibility to be skewness. We re-define the previous parameter to maximum availability and include a skewness factor according to the Zipf-distribution. In our churn traces, we skew peer availability based on the exponent in Equation 9.1.

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)} \tag{9.1}$$

Based on these churn parameters we have defined three churn experiments. To evaluate how our protocol works in rapidly changing environments we have set mean offline time to 8 hours and maximum availability to 75%. We believe these parameters are a fit representation of a dynamic environments such as Skype[36] where peers are likely to participate in the system during work hours.

We have also defined a less dynamic churn trace. Our second trace has a mean offline time of 4 hours and maximum availability of 90%.The availability is skewed based on the distribution mentioned previously. This

trace can represent a system where peers leave the system for short periods of time while still available for 90% of the time.

We use a third churn trace to represent the events that my occur in a more stable and yet dynamic systems. In this trace mean offline time is set to 12 hours while availability is set to 75% as that is similar to peer availability in Skype.

Finally, a summary of the churn experiments are shown in Table 9.3. In all of our churn experiments the parameters will be set to the default values described earlier in Section 9.5. Our churn traces are not used to indicate when peers have left the system, but rather when they have been detected as offline. The difference between these two events depends on the detection speed of the failure detector implemented. For fairness sake, we assume that our baseline uses the same failure detector with the same parameters. This helps our work to be orthogonal from any implementations of failure detectors and focus on the effects of churn on our protocol.

Churn experiments	Max. peer availability	Skewed availability	Mean offline time
Experiment 1	75%	No	8 hrs
Experiment 2	50%	No	4 hrs
Experiment 3	75%	Yes	12 hrs

Table 9.3: Table of the churn experiments and the parameters to describe them.

Chapter 10

Analysis

In this chapter we will provide a sensitivity analysis for some of the parameters we mentioned in Section 9.5. As our replication policy randomly selects the *coordinator* and places replicas at random peers based on a DHT-lookup, it is possible that the seed used to generate the random numbers is a *lucky seed*. For this reason we have simulated each experiment ten time with ten different pseudo-random generator seeds. The seeds were chosen randomly as a number between 0 and 100000 by using "True Random Number Service"[1]. These number are claimed to be truly random as "the randomness comes from atmospheric noise". Due to the different results, we will show error bars for our default parameters. However, for the sake of making figures clear and easy to understand, we will omit the error bars for the rest of the experiments. The results for the rest of the experiments are either median or maximum result of these ten experiment. We will provide information for each of the graphs individually.

10.1 Default values

In this section we will see the results of our default values for the parameters described earlier in Section 9.5. As mentioned before our simulator is deterministic and therefore each experiment leads to the same results given that the parameter of the experiments are the same. However, since our replication policy uses a pseudo-random number generator to decide the *coordinator* for the propagation, we run each experiment with 10 different seeds. Pseudo-random number generators take a seed to generate a list of numbers that appear to be random. If the seed is the same, the pseudo-random number generator will yield the same numbers that appear random. By changing the seed, the pseudo-random number generator will return different numbers.

In Figure 10.1 we see the results of ten different seeds fed into the random generator. The graph shows the median delay for experiments run with the default parameters. Error bars show the maximum and minimum propagation time for propagating 7 copies of files within the

[1] www.random.org

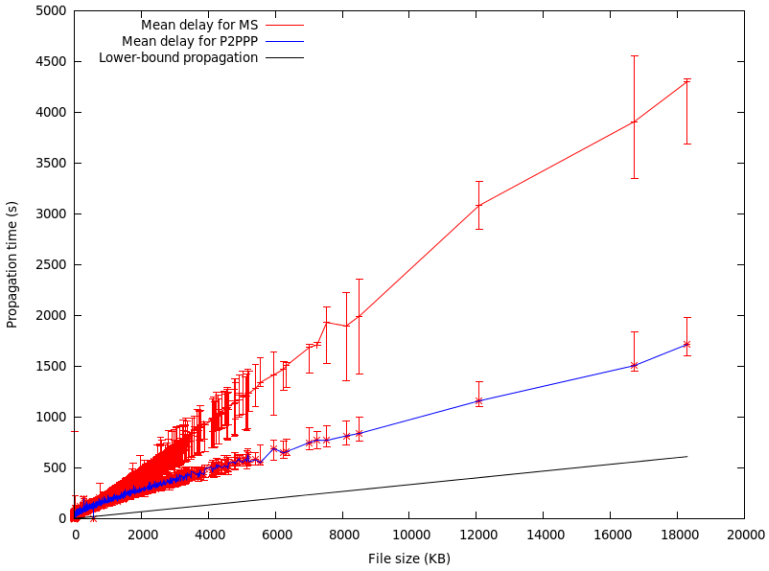

Figure 10.1: The propagation time for proposed default values.

DHT. Obviously as the file sizes increase so does the propagation time. Initially, we expected the delays to be ”smooth” and linear to file sizes. However, the propagation time has some spikes. These spikes in the graph are mostly visible in the MS-protocol. This is due to the event distribution discussed in Section 9.3. In other words, due to the Wikipedia dataset, files inserted into the DHT are not evenly timed. This can lead to multiple propagations starting, possibly with the same peers participating in the propagation. To confirm this, we have run a random event trace, where the same files are inserted into the DHT at a random time. Our results confirmed that, modifying the creation time in our dataset will remove the spikes, thus the Wikipedia event distribution cause the spikes in the graph. The delay graphs resembled a more linear graph than Figure 10.1. As we are using the same dataset for all our experiment, similar spikes will appear during in some other experiments.

Most interesting is the fact that our protocol is more ”stable” in terms of delays. This is due to the load-balancing problem that occurs when the (randomly) chosen *coordinator* is not fit for propagating the file due to other propagations it might be taking part in. Since our protocol reduces the required load from the *coordinator*, choosing a random or unsuitable *coordinator* does not affect the propagation time as much.

Our second evaluation metric is the overhead our protocol introduces to the DHT. The way we measure overhead was previously described in Section 9.2.2 on page 66. Figure 10.2 shows the median push bandwidth consumed by P2PPP. As can be seen in the figure the maximum amount of bandwidth consumed is less than 250 KB, occurred when propagating a file larger than 18 MB. This means that our protocol consumed 250 KB when propagating 126 MB (7 copies of a 18 MB file). The overhead relative to file size is less than 0.2% for the largest file propagated. The small spikes are

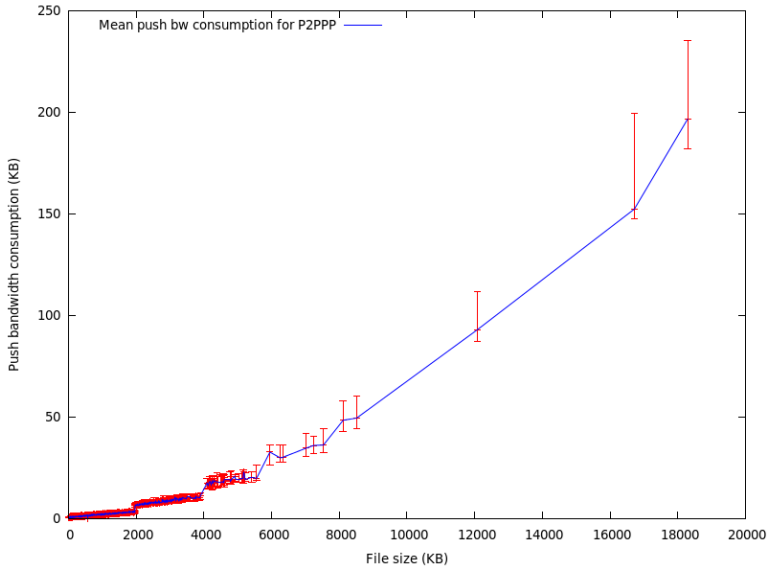

Figure 10.2: Push bandwidth consumed by our protocol.

due to the increases in propagation time. Increase in propagation time leads to an increase in the number of push messages sent and thus the overhead. The small step-wise increase is due to increased number of bitmap that are needed to represent a file in the push messages. In our implementation, we have used 64-bit integers to represent a file. Using a smaller integers to represent pieces that are received, will lead to the same steps; however, there will be many more steps. As the size of the integers used to represent received pieces decrease, we will see a ”smoother” increase and the step-wise increase will disappear. In the next sections we will see how the overhead is affected as some of the parameters are modified.

10.2 Transfer slots

As the number of slots increase so does the parallelism of the protocol. In such cases interested peers can request multiple pieces from their neighbours simultaneously. This leads to decrease in propagation time. Figure 10.3 shows the *maximum* propagation time for propagating 7 copies of each file in the dataset when there are 1, 2, 4, 6 and 8 transfer slots for neighbours. For comparison sake, we have included the propagation delay for the standard master-slave protocol.

As expected, as the number of transfer slots increase, the propagation time decreases. Requesting multiple pieces from a single neighbour can lead to a neighbour that is overloaded with requests. As the Figure 10.3 shows, by having 6 transfer slots for each neighbour, we can utilise neighbour's bandwidth best. However, as the transfer slots for each neighbour increase more than 6 slots, the propagation time decreases. This is because of the policy used to request pieces from neighbours. Our policy for requesting pieces from neighbours is done by iterating through all

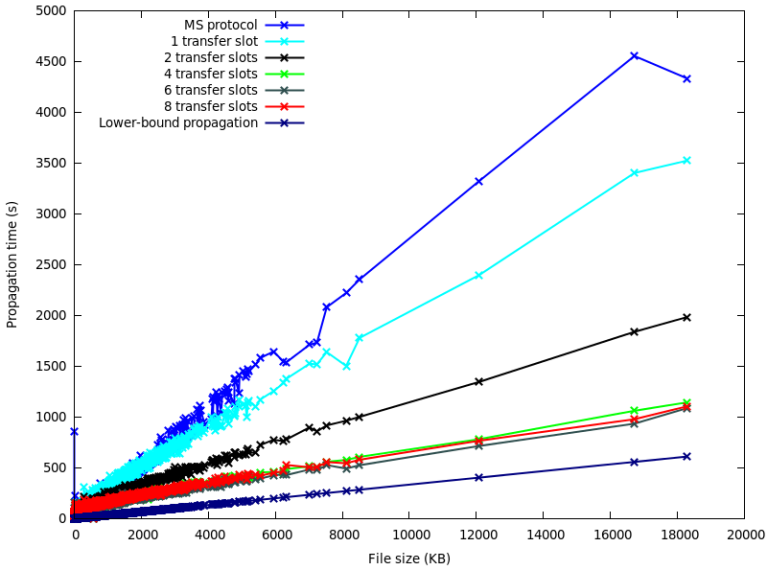

Figure 10.3: Effect of transfer slots on the propagation time.

items, all neighbours interested in that item, and all transfer slots for that neighbour. This can cause an imbalance for certain neighbour that receive multiple pull-requests simultaneously. By having a more dynamic policy it is possible to distribute pull-requests among neighbours evenly. In theory, a load-balanced policy can improve the propagation time by utilising the bandwidth for *all* the participants effectively. However, implementing such policies can be very application-specific.

In short, in our dataset having 6 transfer slots yields the best propagation time. However, the difference between having 4, 6 and 8 transfer slots is minor. Also interesting is the fact that by having only *one* transfer slot for each neighbour, we can reduce the propagation time significantly.

Figure 10.4 shows the median of push bandwidth consumed (for each file) during our propagation. The figure shows that the protocol overhead decreases as the number of slots increase. This is due to the fact that propagation is completed quicker and the number of push messages sent to participants are less. Figure 10.4 shows that having 6 transfer slots consumes less overhead than the alternatives. These results reflect the result seen in Figure 10.3 as overhead consumption is related to propagation time. The most overhead consumed was when there was a single transfer slot for each neighbour i.e. less than 350 KB when propagating the largest file in the dataset. The overhead relative to the file size is less than 0.3%.

As discussed earlier in Section 10.1, there is a step-wise increase in the overhead consumption, the increase is larger when there is only one transfer slot. This is due to the increase in propagation time.

There is a stepwise increase in overhead consumption when propagating files larger than 2000 KB and 4000 KB. This is due to the way we have implemented bitmaps. Our implementation uses an array of 64-bit inte-

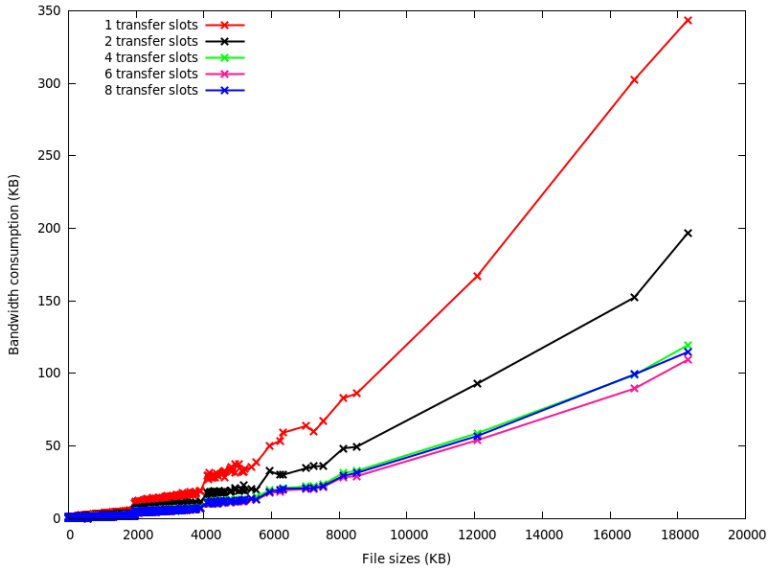

Figure 10.4: Effect of transfer slots on protocol overhead.

gers to represent the state of received pieces. As files that are propagated become larger than 1920 KB (64 pieces that are 30 KB each), we need to include an extra bitmap in the push messages to represent the received pieces. These steps in the graphs can be avoided by implementing smaller sizes integers e.g. 32 or 16-bit integers.

Increase in propagation delay leads to increased number of push iterations that are executed during the propagation. This results in a linear increase in overhead consumption. However, the size of push messages also becomes larger as file sizes increase. The overhead consumption grows in a polynomial manner.

10.3 Piece size

In this section we will see how changes to piece sizes will influence the propagation time in our system. We will use default values and only change the piece size parameter. Figure 10.5 shows the median propagation time as the size of each piece increase. We can see that small pieces increase propagation time. For our dataset, the propagation time is lowest when pieces are 100 KB in size. The propagation time *increases* when pieces are 150 KB. This is due to the fact that neighbours must exchange large pieces and offer more bandwidth for each piece. This leads to some neighbours providing most of the pieces of a file. In theory, as size if pieces increase, the bandwidth that has to be offered becomes less distributed among participants and more centralised.

Figure 10.6 shows the median bandwidth consumed by push messages sent during the propagation. When having 10 KB pieces, the overhead is drastically increased to approx. 1400 KB when propagating the largest file in our dataset. The relative overhead consumption is approximately 1.1%

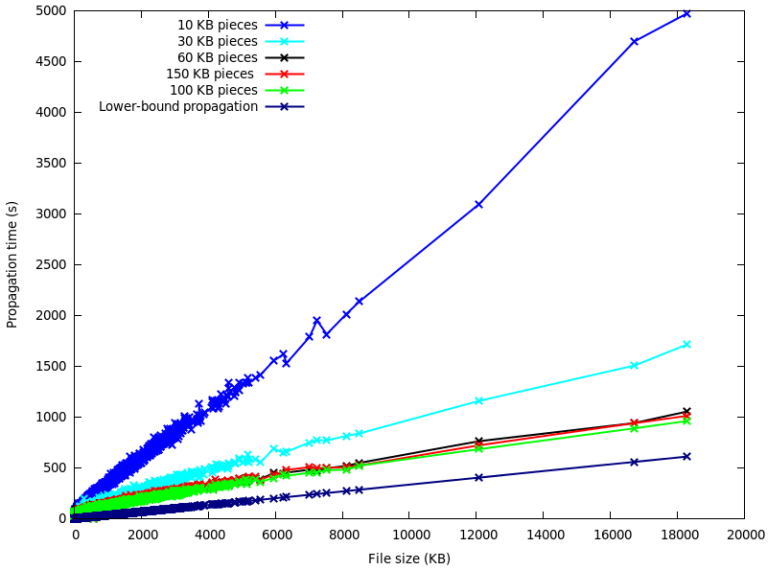

Figure 10.5: Propagation delay for different piece sizes.

relative to the amount of data that is propagated. We have omitted the overhead result of the 10 KB experiment in order to improve the scale of the graph. As can be seen in Figure 10.6, the overhead increases for smaller pieces. Moreover, as piece sizes increase, the overhead consumption looks ”smoother” and the number of ”steps” have decreased due to the size of each piece. Large piece sizes decrease the overhead consumption, especially as the file sizes increase.

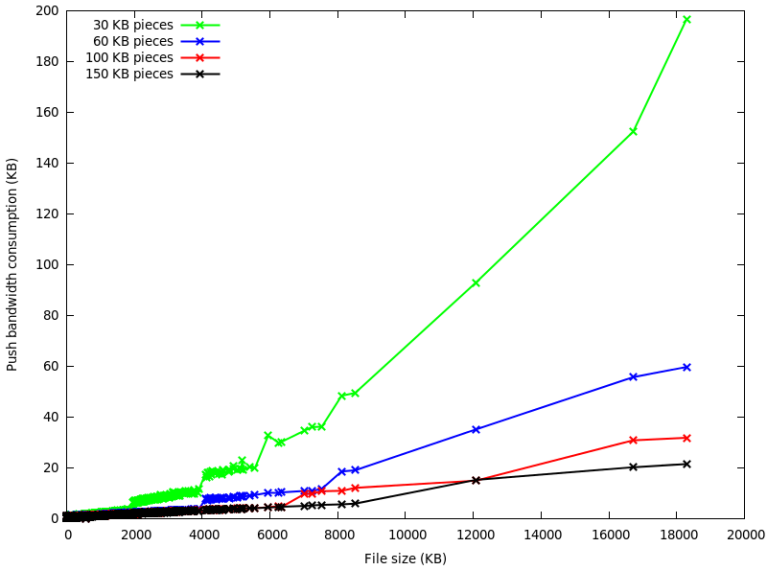

Figure 10.6: Median push bandwidth consumption for different piece sizes.

10.4 Number of replicas

In this section we will change the number of replicas created. The other parameters are set to the default values. We will see how the propagation time and overhead are influenced as the number of files propagated changes. The replication policy we described earlier in Section 5.1.2. Figure 10.7 shows the median delay for propagating files to 5, 7 and 10 peers within the DHT, respectively. However, in order to propagate 10 copies of a file, we need to increase the size of peers returned when looking up in the DHT. To do this, we have chosen to increase the number of returned peers as shown in Table 10.1.

Experiment	Size of DHT-lookup	Replicas
5 replicas	7	5
7 replicas	10	7
10 replicas	13	10

Table 10.1: Values used by the replication policy for choosing coordinators and replicas.

Interestingly, as the number of participants increase, the propagation time increases linearly for P2PPP. This is due to the fact that, the *coordinator* is not forced to do the heavy lifting in the propagation process. Because the propagation load is distributed among all participant peers evenly, the propagation delay grows linearly with file sizes. This implies that the load of propagation is not on a single peer, but rather distributed in the propagation process.

An interesting observation is when 5, 7 and 10 copies of large files are propagated. When propagating large files, the mapping of items in the DHT becomes more important. For instance, if a large file is mapped into a ”busy” place in the DHT, then the propagation will take longer time. We have run some initial tests and confirmed that by increasing the number of peers that are returned in a DHT-lookup, most of these spikes disappear and the propagation time resembles a more linear graph. Increasing the randomness of participants in the propagation, leads to better results. Due to time constraints and scope of our evaluation, we will not run sensitivity analysis on the size of DHT-lookups.

Obviously, the standard master-slave protocol performs worse as the number of participants increase. This is due to the fact that the *coordinator* has to propagation entire files to an increasing number of peers, which increases the propagation time.

As can be seen in Figure 10.7, when propagating 10 copies of a file, delays have more spikes compared to the scenarios where 7 or 5 copies were propagated. As number of participants in the propagation increase, so does the probability that at least one of these peers are also participating in other propagations. The result of which is the spikes seen the graph.

Figure 10.8 shows the median overhead consumption for simulation where the number of participants in the propagation are increased. The

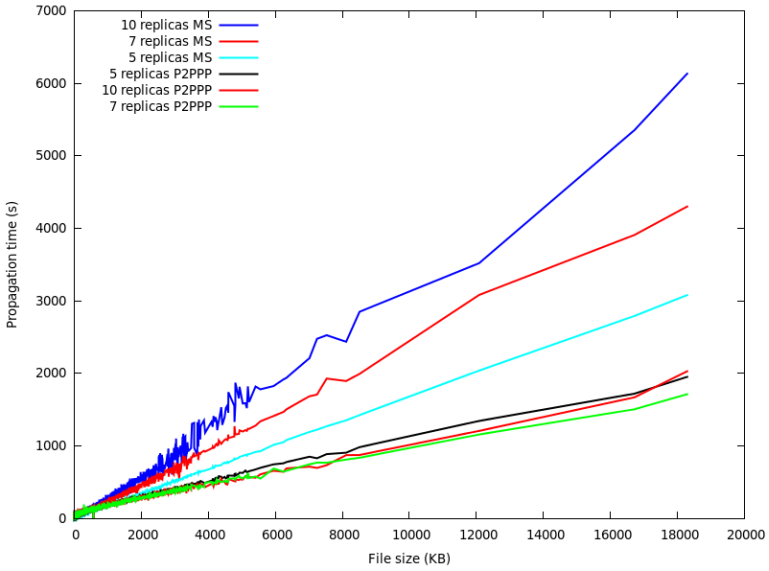

Figure 10.7: Propagation delay when creating 5, 7 and 10 replicas.

overhead is greatest when there are 10 participants. In that case, the largest amount of overhead was consumed when propagating 10 copies of the largest file in our dataset. The total overhead was less than 300 KB. The overhead is less than 0.2% compared to the data being propagated[2]. The small spikes in the overhead are due to the spikes in propagation time. Spikes in the propagation time are due to the distribution of events in Wikipedia dataset.

10.5 Bandwidth

In this section we will *not* return to our default values described earlier in Section 9.5 on page 70. Because choosing piece sizes and number of transfer slots has to be done according to the amount of bandwidth peer contribute, we have modified these two parameters. Table 10.2 shows the changed parameter during our bandwidth analysis experiments. Piece size parameters were chosen solely based on the amount of bandwidth contributed. They were deliberately chosen to take one or two seconds of data transmission. Transfer slots were increased to utilise the increasing amount of bandwidth available, while also utilising the amount of bandwidth neighbours are contributing.

Figure 10.9 shows the propagation time as the peers contribute more bandwidth. The graphs included in the figure relate to both P2PPP and the standard master-slave protocol. To improve the scale of the graph, we have omitted the results of standard master-slave protocol when peers have 15 KB/s and 30 KB/s. From the figure we can see that collaboratively propagating files is much faster. Most interesting is the fact

[2]The size of data being propagated can be calculated by multiplying file size with number of interested peers.

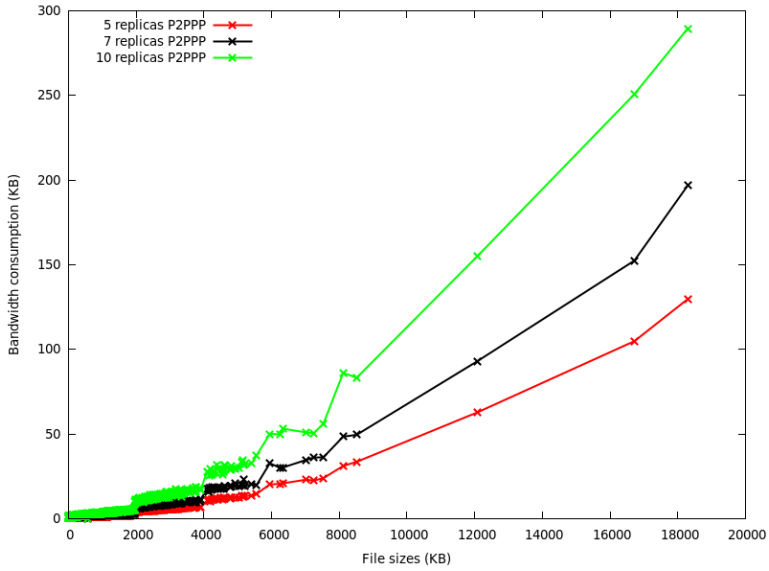

Figure 10.8: Overhead consumption for increasing number of participants in P2PPP protocol.

Experiment	Piece size	Transfer slots
15 KB/s	30 KB	2 slots
30 KB/s (default)	30 KB (default)	2 slots (default)
50 KB/s	50 KB	4 slots
75 KB/s	150 KB	4 slots

Table 10.2: Piece sizes and transfer slots during our analysis of bandwidth sensitivity.

that collaborative propagation with only 15 KB/s bandwidth is close to master-slave propagation when peers have 75 KB/s and definitely faster than 50 KB/s.

During the sensitivity analysis of bandwidth contribution, we have seen that our default values for piece sizes and transfer slots are a better fit for 15 KB/s contribution. For optimising the 30 KB/s bandwidth contribution experiments, these parameter values can to be increased.

Figure 10.10 shows the overhead consumed by push-messages as the bandwidth constraints change in our system. We can see that the overhead consumption is lowest when peers contribute more bandwidth and more there is more bandwidth to utilise. This causes the propagation time to decrease which then effects the overhead consumption.

10.6 Churn results

We have previously described the churn experiment parameters in Section 9.6. In this section we will see the results of our these experiments and interpret the results. The current definition of propagation end is the

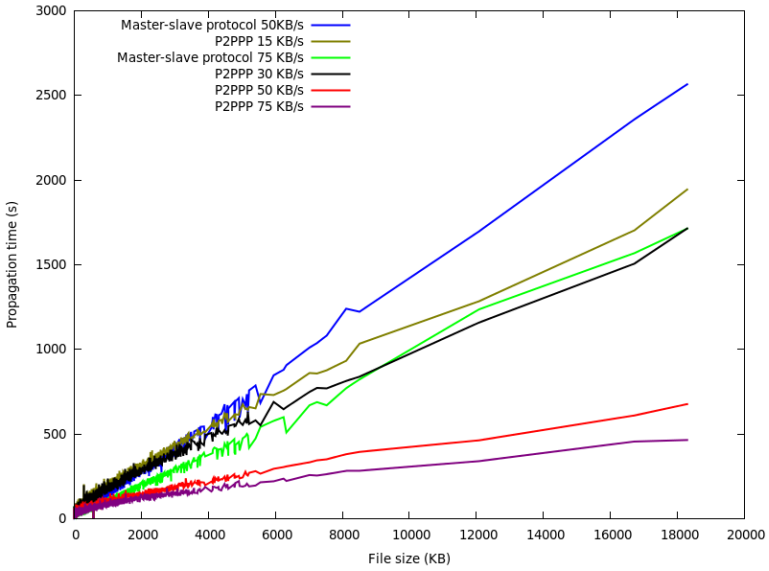

Figure 10.9: Propagation time as peer bandwidth contribution change.

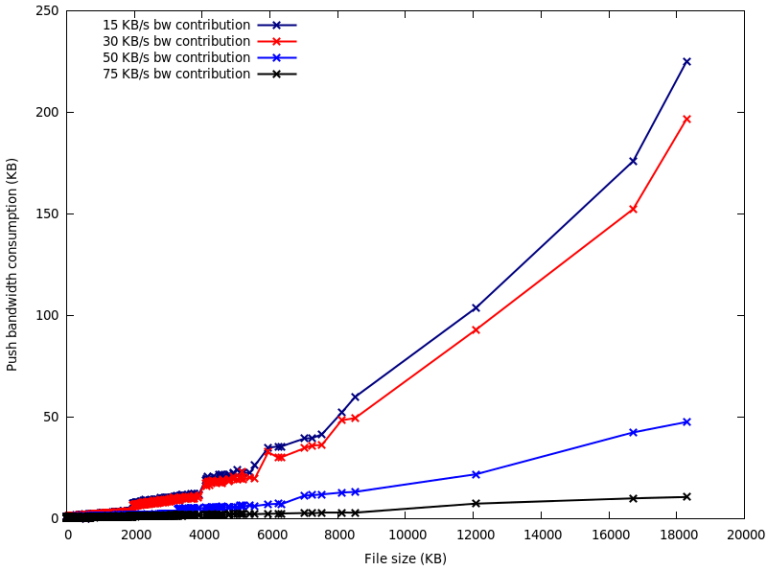

Figure 10.10: Effect of bandwidth contribution on overhead.

time where all peers have the full copy of the file. Due to churn peers could leave the system and join the system later. This leads to abnormal propagation delays as the propagation does not stop before all peers have a copy. To improve the readability and clarity of figures, we will show the propagation time for a median of peers having a full copy. For instance, in our default parameters we propagate 7 copies of a file to 7 different peers. A participant going offline will delay the propagation time drastically, especially if the mean offline time is high. Instead to measuring time it takes for the propagation to stop i.e. 7 copies, we measure the time it takes for a median of peer receiving a full copy i.e. 4 peers. By doing so we can

improve readability and gain insight into the events during propagation, rather at the end of propagation.

The propagation delay illustrated in Figure 10.11 shows the median propagation time, of ten different simulation, when four[3] peers in the DHT have received a full copy of the file being propagated. However, there is still the possiblity that multiple peers that are involved, leave during the propagation. The spikes in the churn experiments can be caused by multiple peers leaving the DHT during propagation. This leads to an increased propagation time for two reasons. First, peers leaving during the propagation can lead to pieces that are only the *coordinator* has. This leads to more peers requesting from the same peer e.g. the coordinator. Secondly, as peers leave during propagation, the propagation load has to be distributed among the remaining peers. Moreover, less peer-to-peer communication and more peer-to-coordinator communication leads to the spikes seen in the graphs.

As can be seen in Figure 10.11, in our 2nd churn experiment the propagation delay decreased when propagating a file approx. 16 MB in size. We speculate that this is due to the results of the DHT-lookup. The DHT-lookup in the simulator only return peers that are alive at the time of lookup. For instance, peer p can participate in multiple propagation and leads to delay in some of the propagations it is part of. If p goes offline, other propagation may propagate faster due to the removed propagation bottleneck. Unfortunately, due to our time-constraints this remains a speculation.

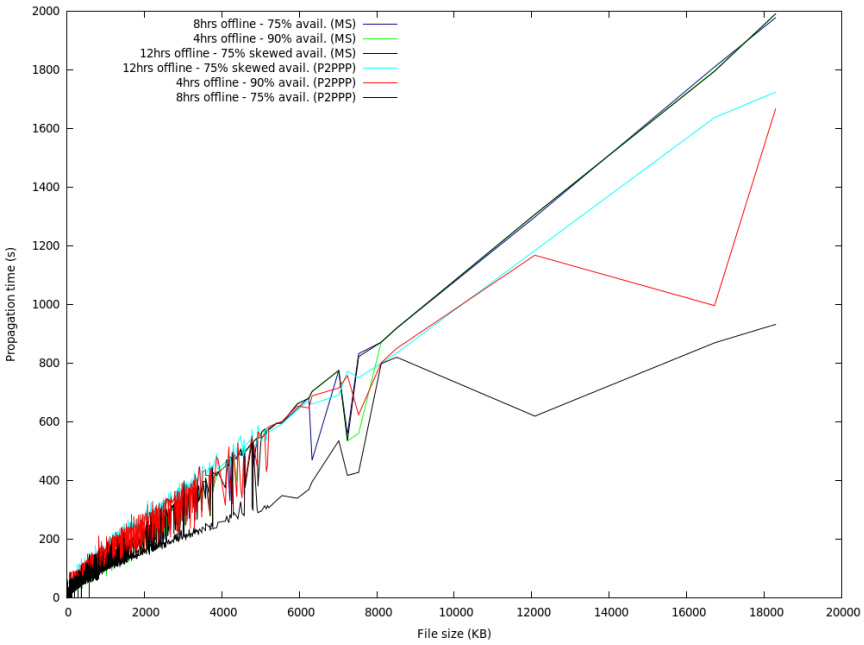

Figure 10.11: Median propagation time of all churn experiments.

Figure 10.12 shows the overhead consumed by extra push messages

[3]Median of our default value for replicas.

during the propagation period. We have included the overhead consumption of all churn experiments. Although we had to modify the propagation delay, we have not modified the way me measure overhead consumption. Figure 10.12 shows the push bandwidth consumption for propagating 7 copies of all files in the system. Obviously spikes in the propagation time are reflected in the overhead graphs as well. However, in our 1st churn experiment, we can see that churn impacts the large files drastically. We speculate that the reason for this impact is the large size of the file, the size of the push messages and the peers that went offline during the propagation of that file. Unfortunately, investigating this issue is time consuming. However, it is notable that for the churn experiment with skewed availability, the overhead consumption are almost identical.

Nevertheless the most overhead consumption encountered during our evaluation was in these experiments. The largest consumption was 700 KB for propagating 7 copies of 18.2 MB file. The overhead relative to the amount of data that was propagated was less than 0.6%.

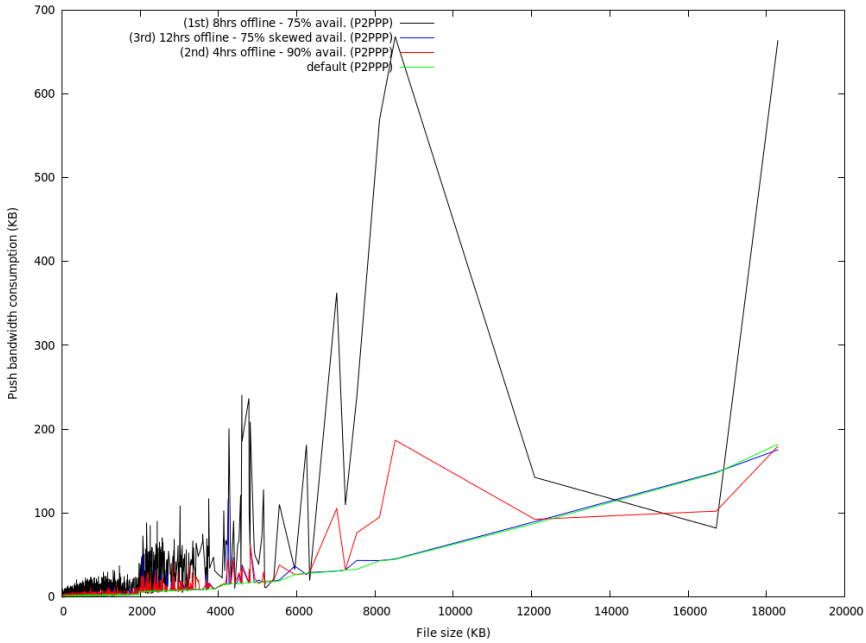

Figure 10.12: Overhead consumption in churn experiments.

10.7 Summary

In this chapter we have analysed the influence of the essential parameters in our system. We gained knowledge of how these parameters effect the propagation protocol and our evaluation metrics. In addition, we have discussed what factors that help reduce propagation time or delay.Factors such as pieces size, number of transfer slots, peer bandwidth, frequency of push messages, churn and the number of replicas were discussed and evaluated. We emphasise that our churn trace implied the detection of

failure, not occurrence of failure. Therefore, participants in P2PPP did not send a request to such offline peers.

Chapter 11

Conclusion and Discussion

In this chapter we will conclude our work and discuss some of the lessons that were learned. But first we will discuss the results of the experiments and the factors that need to be considered for each parameter.

11.1 Propagation time

For our dataset, there were some spikes in the propagation delay. During our evaluation we have confirmed that these spikes are due to the frequency of image uploads. For instance, images are more likely to be uploaded in day-time rather than night-time. If the images uploaded in the same time were mapped to the same area in the DHT, the propagation delay for that file would increase abnormally. We have confirmed that this is the case, by introducing the same files at a random time. In that experiment there were no spikes in the propagation time.

To avoid the spikes in the propagation load, items can be mapped to peers differently. Another solution is to implement a different replication policy to place replicas completely random, instead of our current implementation which is based on a DHT-lookup. A third possible solution is to change the policy by which peers pull pieces. By prioritising ”fast” peers, the propagation delay might decrease.

The obvious factors to consider are piece size and transfer slots. But to utilise the bandwidth efficiently, peer bandwidth must also be considered. The simple technique to use is to let piece size and transfer slots be close to the amount of ”download” bandwidth contributed by peers.

By utilising the entire bandwidth, pulling a file is not longer the bottleneck. In the initial phase of propagation replicas request pieces from the *coordinator* without knowledge of each other. Every piece received in this stage has been sent from the *coordinator*. The frequency of push messages helps reduce the duration of the initial phase. In order to spread this knowledge quickly, push procedures must be executed more often. However, sending frequent push messages outside the propagation period can lead to a lot of overhead. It is possible to reduce this overhead by dynamically changing the frequency of push messages and pushing less frequently when there is no need. Moreover, to limit the overhead

introduced, the dynamically choosing push frequency can be done only for the peers participating in the propagation.

11.2 Overhead

During our evaluation we have realised that both propagation time and the size of bitmaps are relevant for overhead consumption. In other words, the factors that are important to consider for the overhead metric are piece sizes, push frequency and peer bandwidth. Pieces sizes are important to consider as it takes significant amount of maps to represent a large file with small pieces. For instance, a file with 1 GB should not be divided into 10 KB pieces as it would take a lot of maps to represent that file. Limiting the number of maps included in push messages is essential to reducing the overhead as it reduces the size of push messages. Another factor is the frequency of push-messages. To reduce overhead, push frequency might also have to be reduced. Although this contradicts what we discussed in the previous section, there should be a threshold for push frequency to avoid wasting bandwidth while reducing the propagation time. Another possibility is to create a dynamic push frequency where the frequency of push messages increase during propagation. Possible factors that need to be considered are the bandwidth capacity of peers and their neighbours, size of files and number of participants in the propagation.

11.3 Future work

An interesting functionality to see can be an algorithm to choose piece sizes dynamically. Variables such as average peer bandwidth for peers that are participating in the propagation and possibly network congestion could be considered in this algorithm. To improve the load-balancing, it is possible to create an algorithm to prioritise pulling pieces from participants rather than the *coordinator*. However there could be some circumstances where it a better to pull from the *coordinator* e.g. faster response time.

As this protocol can be very useful when propagating multimedia content, specific file and piece selection policies could be implemented to provide a quality of service better suited for users applications. For instance, streaming services could have a sequential piece policy where pieces are requested sequentially.

A real-world implementation of our propagation protocol within a DHT might give insight into how well P2PPP performs in a real-world implementation and the improvement introduced in terms of load-balancing in the DHTs. In addition, it would be interesting to create a new baseline for evaluation where the propagation is done using network coding [33] and see how P2PPP performs compared to network coding and which is better suited within a DHT.

Bibliography

[1] Cisco Systems Inc. URL: http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html (visited on 01/07/2013).

[2] *Apache Cassandra*. The Apache Software Foundation. URL: http://cassandra.apache.org/ (visited on 01/05/2013).

[3] Matthias Bender et al. 'MINERVA: collaborative P2P search'. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB '05. Trondheim, Norway, 2005, pp. 1263–1266. ISBN: 1-59593-154-6.

[4] Antony Chazapis and Nectarios Koziris. 'Storing and Locating Mutable Data in Structured Peer-to-Peer Overlay Networks'. In: *Advances in Informatics*. Ed. by Panayiotis Bozanis and Elias Houstis. Vol. 3746. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 234–244.

[5] Antony Chazapis and Nectarios Koziris. 'XOROS: A mutable Distributed Hash Table'. In: (2009).

[6] Gregory Chockler et al. 'SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication'. In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. DEBS '07. New York, NY, USA: ACM, 2007, pp. 14–25. ISBN: 978-1-59593-665-3. DOI: http://doi.acm.org/10.1145/1266894.1266899. URL: http://doi.acm.org/10.1145/1266894.1266899.

[7] Ian Clarke et al. 'Freenet: A distributed anonymous information storage and retrieval system'. In: *Designing Privacy Enhancing Technologies*. Springer. 2001, pp. 46–66.

[8] *Content delivery technology*. BitTorrent, Inc. URL: http://www.bittorrent.com/ (visited on 01/05/2013).

[9] Leudo Antonio Cutillo, Refik Molva and Thorsten Strufe. 'Privacy preserving social networking through decentralization'. In: *Wireless On-Demand Network Systems and Services, 2009. WONS 2009. Sixth International Conference on*. IEEE. 2009, pp. 145–152.

[10] Frank Dabek et al. 'Wide-area cooperative storage with CFS'. In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), pp. 202–215. ISSN: 0163-5980. DOI: 10.1145/502059.502054. URL: http://doi.acm.org/10.1145/502059.502054.

[11] Giuseppe DeCandia et al. 'Dynamo: amazon's highly available key-value store'. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[12] Alan Demers et al. 'The Bayou architecture: Support for data sharing among mobile users'. In: *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*. IEEE. 1994, pp. 2–7.

[13] *eMule Project*. eMule project. URL: http://www.emule-project.net/ (visited on 01/05/2013).

[14] P.T. Eugster et al. 'Epidemic information dissemination in distributed systems'. In: *Computer* 37.5 (May 2004), pp. 60 –67. ISSN: 0018-9162. DOI: 10.1109/MC.2004.1297243.

[15] Jim Gray et al. 'The transaction concept: Virtues and limitations'. In: *Proceedings of the Very Large Database Conference*. 1981, pp. 144–154.

[16] Tomas Isdal et al. 'Privacy-preserving P2P data sharing with OneSwarm'. In: *ACM SIGCOMM Computer Communication Review*. Vol. 40. 4. ACM. 2010, pp. 111–122.

[17] Sitaram Iyer, Antony Rowstron and Peter Druschel. 'Squirrel: A decentralized peer-to-peer web cache'. In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM. 2002, pp. 213–222.

[18] David Karger et al. 'Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web'. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[19] *Kazaa*. Kazaa. URL: http://www.kazaa.com/ (visited on 01/05/2013).

[20] G. Kreitz and F. Niemela. 'Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming'. In: *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*. 2010, pp. 1 –10.

[21] John Kubiatowicz et al. 'OceanStore: an architecture for global-scale persistent storage'. In: *SIGPLAN Not.* 35 (11 2000), pp. 190–201. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/356989.357007. URL: http://doi.acm.org/10.1145/356989.357007.

[22] Avinash Lakshman and Prashant Malik. 'Cassandra: a decentralized structured storage system'. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[23] Leslie Lamport. 'The part-time parliament'. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[24] Leslie Lamport, Robert Shostak and Marshall Pease. 'The Byzantine generals problem'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[25] Petar Maymounkov and David MaziÃ¨res. 'Kademlia: A Peer-to-Peer Information System Based on the XOR Metric'. In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek and Antony Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-44179-3.

[26] Athicha Muthitacharoen et al. *Etna: a fault-tolerant algorithm for atomic mutable dht data*. Tech. rep. 2005.

[27] Athicha Muthitacharoen et al. 'Ivy: a read/write peer-to-peer file system'. In: *SIGOPS Oper. Syst. Rev.* 36 (SI Dec. 2002), pp. 31–44. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/844128.844132. URL: http://doi.acm.org/10.1145/844128.844132.

[28] Satoshi Nakamoto. 'Bitcoin: A peer-to-peer electronic cash system'. In: *Consulted* 1 (2008), p. 2012.

[29] Alexandra Olteanu and Guillaume Pierre. 'Towards robust and scalable peer-to-peer social networks'. In: *Proceedings of the Fifth Workshop on Social Network Systems*. ACM. 2012, p. 10.

[30] *PeerSim P2P Simulator*. None. URL: http://peersim.sourceforge.net/ (visited on 01/05/2013).

[31] Sylvia Ratnasamy et al. 'A scalable content-addressable network'. In: *SIGCOMM Comput. Commun. Rev.* 31 (4 2001), pp. 161–172. ISSN: 0146-4833. DOI: http://doi.acm.org/10.1145/964723.383072. URL: http://doi.acm.org/10.1145/964723.383072.

[32] M. Ripeanu. 'Peer-to-peer architecture case study: Gnutella network'. In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. 2001, pp. 99 –100. DOI: 10.1109/P2P.2001.990433.

[33] Rodrigo Rodrigues and Barbara Liskov. 'High availability in DHTs: Erasure coding vs. replication'. In: *Peer-to-Peer Systems IV*. Springer, 2005, pp. 226–239.

[34] Antony Rowstron and Peter Druschel. 'Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems'. In: *Lecture Notes in Computer Science* 2218 (2001), 329–?? URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.5987.

[35] Yasushi Saito and Marc Shapiro. 'Optimistic replication'. In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81.

[36] *Skype Communications*. Skype. URL: http://www.skype.com/en/ (visited on 01/05/2013).

[37] *Spotify*. Spotify Sweden AB. URL: https://www.spotify.com/ (visited on 01/05/2013).

[38] C.A. Stein, M.J. Tucker and M.I. Seltzer. 'Building a reliable mutable file system on peer-to-peer storage'. In: *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. 2002, pp. 324 –329. DOI: 10.1109/RELDIS.2002.1180204.

[39] Ion Stoica et al. 'Chord: A scalable peer-to-peer lookup service for internet applications'. In: *SIGCOMM Comput. Commun. Rev.* 31 (4 2001), pp. 149–160. ISSN: 0146-4833. DOI: http://doi.acm.org/10.1145/964723.383071. URL: http://doi.acm.org/10.1145/964723.383071.

[40] Guido Urdaneta, Guillaume Pierre and Maarten Van Steen. 'A survey of DHT security techniques'. In: *ACM Comput. Surv.* 43.2 (Feb. 2011), 8:1–8:49. ISSN: 0360-0300. DOI: 10.1145/1883612.1883615. URL: http://doi.acm.org/10.1145/1883612.1883615.

[41] G.A. Urdaneta Paredes. 'Colloborative Wikipedia hosting'. PhD thesis. Amsterdam: Vrije Universiteit, 2011.

[42] Ben Y Zhao et al. 'Tapestry: A resilient global-scale overlay for service deployment'. In: *Selected Areas in Communications, IEEE Journal on* 22.1 (2004), pp. 41–53.